线程模型深度剖析

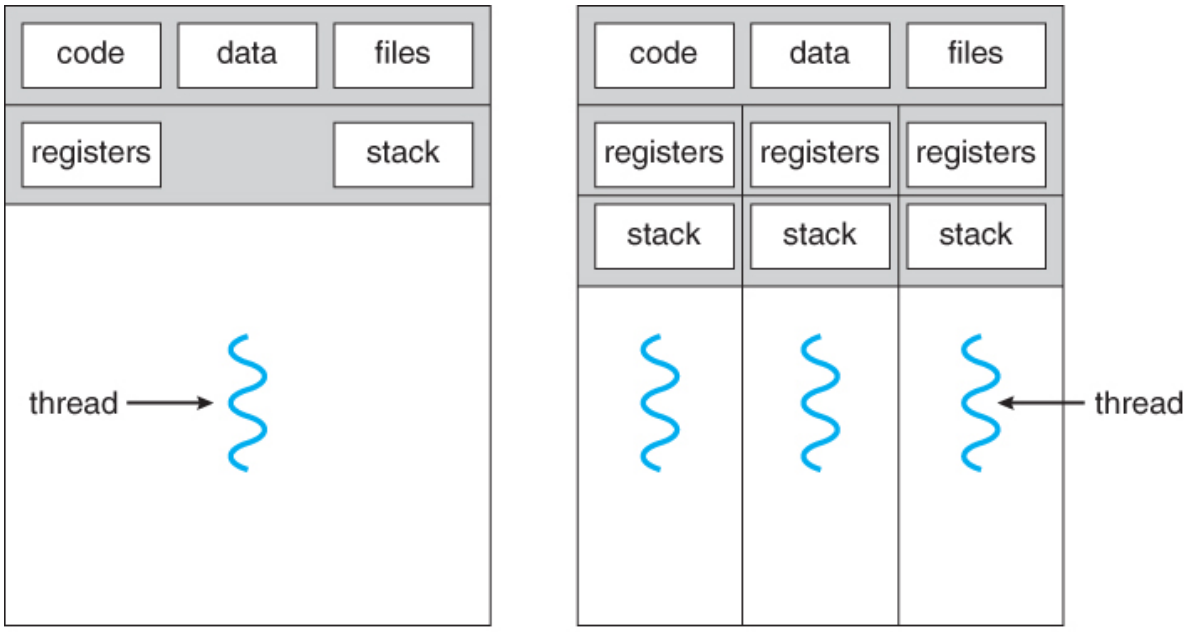
线程

什么是线程？

一个***线程***是CPU利用率的基本单元，包括一个程序计数器，堆栈，一组寄存器和线程ID。

传统（重量级）进程具有单个控制线程 - 有一个程序计数器，以及可在任何给定时间执行的一系列指令。

多线程应用程序在单个进程中具有多个线程，每个线程都有自己的程序计数器，堆栈和寄存器集，但共享公共代码，数据和某些结构（如打开文件）。



single-threaded process　　multithreaded process

多线程有四大类优点

响应性 - 一个线程可以提供快速响应，而其他线程被阻塞或减慢进行密集计算。

资源共享 - 默认情况下，线程共享公共代码，数据和其他资源，这允许在单个地址空间中同时执行多个任务。

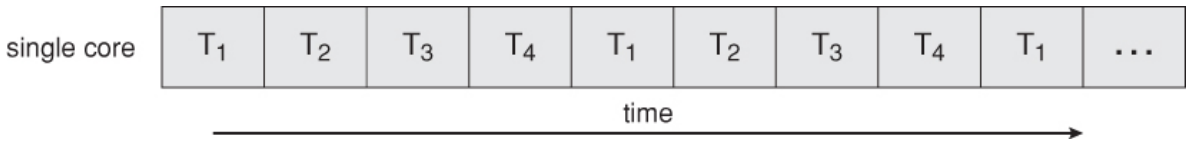
经济 - 创建和管理线程（以及它们之间的上下文切换）比为进程执行相同的任务要快得多。

可伸缩性，即多处理器体系结构的利用 - 单线程进程只能在一个CPU上运行，无论有多少可用，而多线程应用程序的执行可能在可用处理器之间分配。（请注意，当有多个进程争用CPU时，即当负载平均值高于某个特定阈值时，单线程进程仍然可以从多处理器体系结构中受益。）

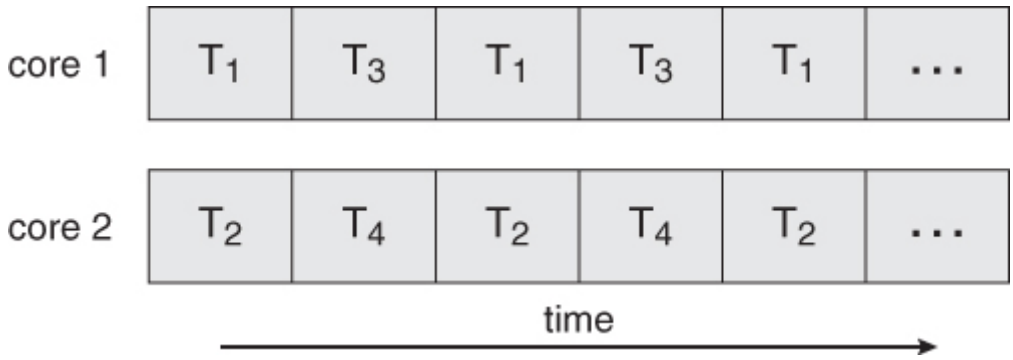
多核编程

计算机体系结构的最新趋势是在单个芯片上生产具有多个***核心***或CPU的芯片。

在传统的单核芯片上运行的多线程应用程序必须交错线程，如下图图所示。



但是，在多核芯片上，线程可以分布在可用内核上，从而实现真正的并行处理，如图所示：



对于操作系统，多核芯片需要新的调度算法以更好地利用可用的多个核。
随着多线程变得越来越普遍和越来越重要（数千而不是数十个线程），CPU已被开发用于支持硬件中每个核心更多的同步线程。

多核芯片的挑战

识别任务 - 检查应用程序以查找可以同时执行的活动。
平衡 - 查找同时运行的任务，提供相同的价值。即不要浪费一些线程来完成琐碎的任务。
数据拆分 - 防止线程相互干扰。
数据依赖性 - 如果一个任务依赖于另一个任务的结果，则需要同步任务以确保以正确的顺序进行访问。
测试和调试 - 在并行处理情况下本身就更加困难，因为竞争条件变得更加复杂和难以识别。

并行类型

从理论上讲，有两种不同的工作负载并行化方法：

数据并行性

在多个核（线程）之间划分数据，并在数据的每个子集上执行相同的任务。例如，将大图像分成多个片段并对不同核心上的每个片段执行相同的数字图像处理。

任务并行性

划分要在不同核心之间执行的不同任务并同时执行它们。

在实践中，任何程序都不会仅仅由这些中的一个或另一个划分，而是**通过某种混合组合**。

线程的类型

内核线程(KLT)

内核线程就是直接由操作系统内核支持的线程，这种线程由内核来完成线程切换，内核通过操纵调度器对线程进行调度，并负责将线程的任务映射到各个处理器上。每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情，支持多线程的内核就叫做多线程内核。

轻量级进程(LWP)

轻量级进程是由系统提供给用户的操作内核线程的接口的实现。即轻量级进程是内核线程的一个替身。

用户线程(UT)

用户线程建立在用户空间的线程库上，系统内核不能感知线程存在的实现。用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助。如果程序实现得当，这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持规模更大的线程数量，部分高性能数据库中的多线程就是由用户线程实现的。

线程模型

有两种不同方法来提供线程支持：用户层的用户线程或内核层的内核线程。

用户线程位于内核之上，它的管理无需内核支持；而内核线程由操作系统来直接支持与管理。几乎所有的现代操作系统，包括 Windows、Linux、Mac OS X 和 Solaris，都支持内核线程。

最终，用户线程和内核线程之间必然存在某种关系。本节研究三种常用的建立这种关系的方法：多对一模型、一对一模型和多对多模型。

多对一模型

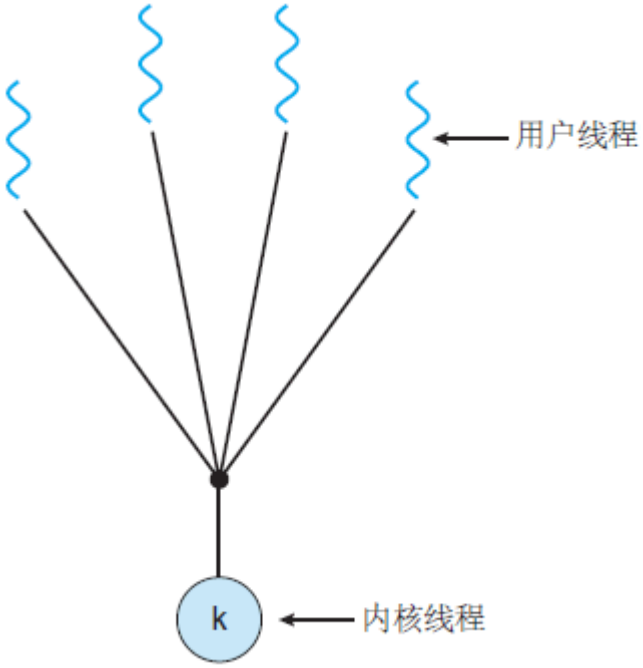


图 1 多对一模型

多对一模型（图 1）映射多个用户级线程到一个内核线程。

线程管理是由用户空间的线程库来完成的，因此效率更高。不过，如果一个线程执行阻塞系统调用，那么整个进程将会阻塞。再者，因为任一时间只有一个线程可以访问内核，所以多个线程不能并行运行在多处理核系统上。

Green threads 线程库为 Solaris 所采用，也为早期版本的 Java 所采纳，它就使用了多对一模型。然而，现在几乎没有系统继续使用这个模型，因为它无法利用多个处理核。

一对一模型

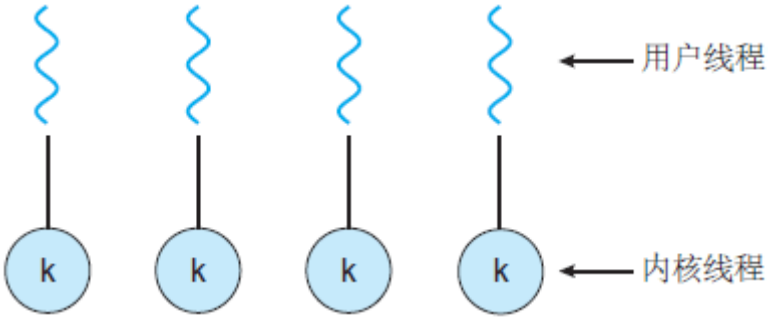


图 2 一对一模型

一对一模型（图 2）映射每个用户线程到一个内核线程。

该模型在一个线程执行阻塞系统调用时，能够允许另一个线程继续执行，所以它提供了比多对一模型更好的并发功能；它也允许多个线程并行运行在多处理器系统上。

这种模型的唯一缺点是，创建一个用户线程就要创建一个相应的内核线程。由于创建内核线程的开销会影响应用程序的性能，所以这种模型的大多数实现限制了系统支持的线程数量。Linux，还有 Windows 操作系统的家族，都实现了一对一模型。

多对多模型

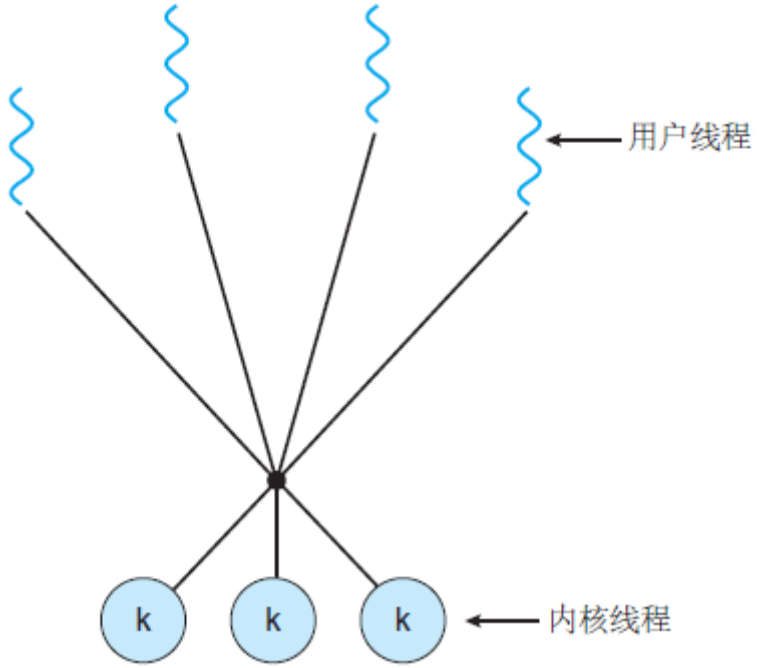


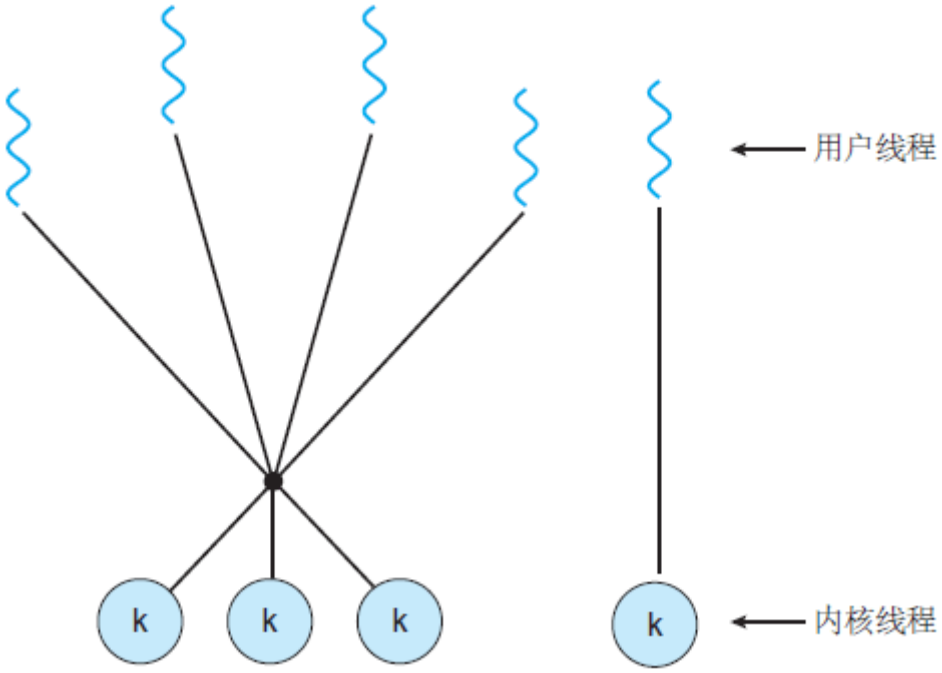
图 3 多对多模型

多对多模型（图 3）多路复用多个用户级线程到同样数量或更少数量的内核线程。内核线程的数量可能与特定应用程序或特定机器有关（应用程序在多处理器上比在单处理器上可能分配到更多数量的线程）。

现在我们考虑一下这些设计对并发性的影响。虽然多对一模型允许开发人员创建任意多的用户线程，但是由于内核只能一次调度一个线程，所以并未增加并发性。虽然一对一模型提供了更大的并发性，但是开发人员应小心，不要在应用程序内创建太多线程（有时系统可能会限制创建线程的数量）。

多对多模型没有这两个缺点：开发人员可以创建任意多的用户线程，并且相应内核线程能在多处理器系统上并发执行。而且，当一个线程执行阻塞系统调用时，内核可以调度另一个线程来执行。

多对多模型的一种变种仍然多路复用多个用户级线程到同样数量或更少数量的内核线程，但也允许绑定某个用户线程到一个内核线程。这个变种，有时称为双层模型（图 4）。



参考文献：

https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

https://blog.csdn.net/21aspnet/article/details/89061134

Java线程基础

实现的方式

1. 继承Thread
2. 实现Runnable

Thread的API方法

getName()获取线程名称。
getId()获取唯一标识。
start()是异步的，方法通知线程规划器此线程已经准备就绪，等待调用线程对象的run()方法。
thread.run()是同步的，有main主线程调用run方法，必须等run方法中的代码执行完才能执行后面的代码。
执行start方法的顺序不代表线程启动的顺序。
this:当前线程。
currentThread()：表示承载当前线程的线程。 isAlive()判断当前线程是否处于活动状态。
sleep()当前线程休眠。
interrupt()给当前线程标记一个中断状态。不会中断线程。
interrupted()判断当前线程是否含被标记中断。执行后讲标记状态清除。
isInterrupted()判断当前线程是否被标记中断，但不清除状态标识。
stop()立即停止线程。
suspend()暂停线程。 resume()恢复线程。
yidld()放弃当前cup资源，将它让给其他任务去占用cpu执行时间。但是时间不确定。
setPriority()设置线程的优先级。 1~10级，默认为5
setDaemon(true):守护线程,当进程中不存在用户线程了，则守护线程自动销毁。守护线程创建的线程也是守护线程。

实现Runnable接口创建线程

创建类A实现Runnable，重写run方法。
申明类A对象，Thread thread=new Thread(类A对象);thread.run();

特点

在sleep中的线程被interrupt中断时，会抛出InterruptedException异常
线程调用stop()时，隐式抛出ThreadDeath异常。使用stop()可能给数据造成不一致的结果，是一个过期的方法。
可以使用return结合interrupt使用来控制线程的停止。
suspend和resume使用时会独占对象，使得其他线程无法访问公共同步对象。
suspend()执行后会导致线程不同步，从而导致数据一致性问题。还会影响synchronized方法。
优先级具有随机性。

锁原理

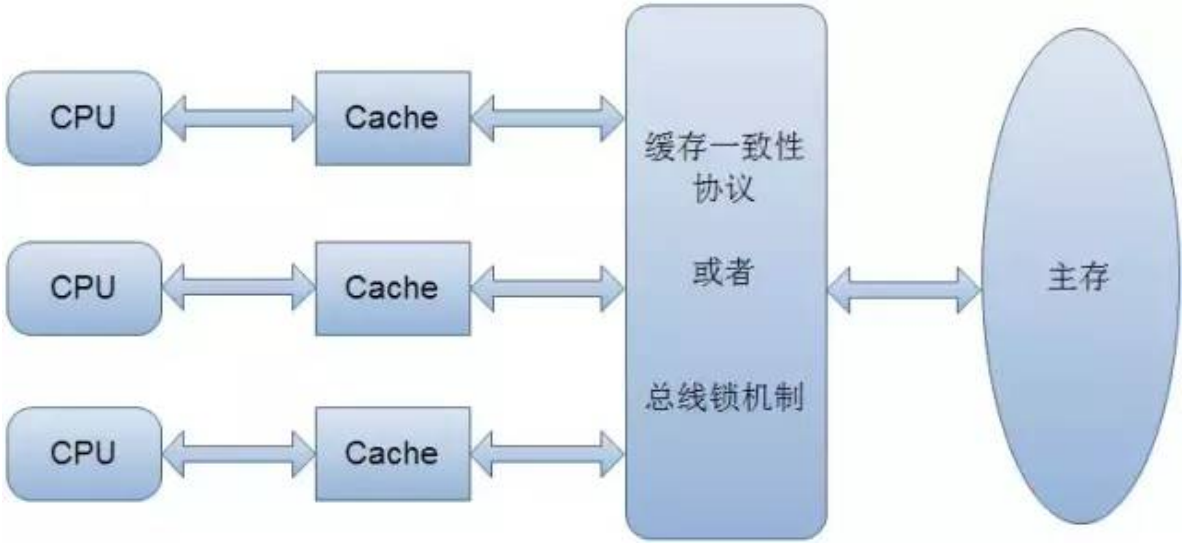
1、为什么要用锁？

锁-是为了解决并发操作引起的脏读、数据不一致的问题。

2、锁实现的基本原理

2.1、volatile

> Java编程语言允许线程访问共享变量， 为了确保共享变量能被准确和一致地更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁要更加方便。
>
> volatile在多处理器开发中保证了共享变量的“ 可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。



结论：如果volatile变量修饰符使用恰当的话，它比synchronized的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。

2.2、synchronized

> synchronized通过锁机制实现同步。

先来看下利用synchronized实现同步的基础：Java中的每一个对象都可以作为锁。

具体表现为以下3种形式。

- 对于普通同步方法，锁是当前实例对象。
- 对于静态同步方法，锁是当前类的Class对象。
- 对于同步方法块，锁是Synchonized括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

2.2.1 synchronized实现原理

> synchronized是基于Monitor来实现同步的。

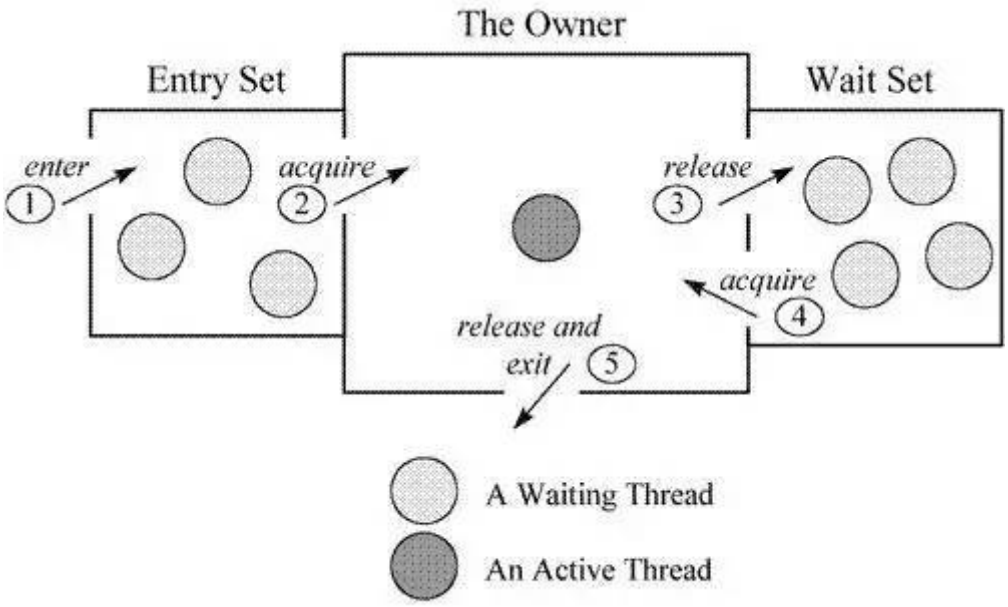
Monitor从两个方面来支持线程之间的同步：

- 互斥执行
- 协作

1、Java 使用对象锁 (使用 synchronized 获得对象锁) 保证工作在共享的数据集上的线程互斥执行。

2、使用 notify/notifyAll/wait 方法来协同不同线程之间的工作。

3、Class和Object都关联了一个Monitor。



- 线程进入同步方法中。
- 为了继续执行临界区代码，线程必须获取 Monitor 锁。如果获取锁成功，将成为该监视者对象的拥有者。任一时刻内，监视者对象只属于一个活动线程（The Owner）
- 拥有监视者对象的线程可以调用 wait() 进入等待集合（Wait Set），同时释放监视锁，进入等待状态。
- 其他线程调用 notify() / notifyAll() 接口唤醒等待集合中的线程，这些等待的线程需要**重新获取监视锁后**才能执行 wait() 之后的代码。
- 同步方法执行完毕了，线程退出临界区，并释放监视锁。

参考文档：https://www.ibm.com/developerworks/cn/java/j-lo-synchronized

2.2.2 synchronized具体实现

1、同步代码块采用monitorenter、monitorexit指令显式的实现。

2、同步方法则使用ACC_SYNCHRONIZED标记符隐式的实现。

通过实例来看看具体实现：

```
public class SynchronizedTest {
   public synchronized void method1(){   System.out.println("Hello World!");   }
   public  void method2(){   synchronized (this){   System.out.println("Hello
World!");   }   }}
```

javap编译后的字节码如下：

```
  public synchronized void method1();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_SYNCHRONIZED//同步方法的实现
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2                  // Field java/lang/System.out:Ljava/io
         3: ldc           #3                  // String Hello World!
         5: invokevirtual #4                  // Method java/io/PrintStream.println:
         8: return
      LineNumberTable:
        line 9: 0
        line 10: 8
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       9     0  this   Lsync/SynchronizedTest;

  public void method2();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=1
         0: aload_0
         1: dup
         2: astore_1
         3: monitorenter//同步代码块的实现
         4: getstatic     #2                  // Field java/lang/System.out:Ljava/io
         7: ldc           #3                  // String Hello World!
         9: invokevirtual #4                  // Method java/io/PrintStream.println:
        12: aload_1
        13: monitorexit//同步代码块的实现
        14: goto          22
        17: astore_2
        18: aload_1
        19: monitorexit
        20: aload_2
        21: athrow
        22: return
      Exception table:
        ...
      LineNumberTable:
        ...略
      LocalVariableTable:
        ...略
      StackMapTable: number_of_entries = 2
        ...略
}
SourceFile: "SynchronizedTest.java"
```

monitorenter

每一个对象都有一个monitor，一个monitor只能被一个线程拥有。当一个线程执行到monitorenter指令时会尝试获取相应对象的monitor，获取规则如下：

- 如果monitor的进入数为0，则该线程可以进入monitor，并将monitor进入数设置为1，该线程即为monitor的拥有者。
- 如果当前线程已经拥有该monitor，只是重新进入，则进入monitor的进入数加1，所以synchronized关键字实现的锁是可重入的锁。
- 如果monitor已被其他线程拥有，则当前线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor。

monitorexit

只有拥有相应对象的monitor的线程才能执行monitorexit指令。每执行一次该指令monitor进入数减1，当进入数为0时当前线程释放monitor，此时其他阻塞的线程将可以尝试获取该monitor。

2.2.3 锁存放的位置

锁标记存放在Java对象头的Mark Word中。

长　度	内　容	说　明
32/64bit	Mark Word	存储对象的 hashCode 或锁信息等
32/64bit	Class Metadata Address	存储到对象类型数据的指针
32/32bit	Array length	数组的长度（如果当前对象是数组）

锁状态	25bit	4bit	1bit 是否是偏向锁	2bit 锁标志位
无锁状态	对象的 hashCode	对象分代年龄	0	01

锁状态	25bit		4bit	1bit	2bit
	23bit	2bit		是否是偏向锁	锁标志位
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向互斥量（重量级锁）的指针				10
GC 标记	空				11
偏向锁	线程 ID	Epoch	对象分代年龄	1	01

锁状态	25bit	31bit	1bit	4bit	1bit	2bit
			cms_free	分代年龄	偏向锁	锁标志位
无锁	unused	hashCode			0	01
偏向锁	ThreadID(54bit) Epoch(2bit)				1	01

2.2.3 synchronized的锁优化

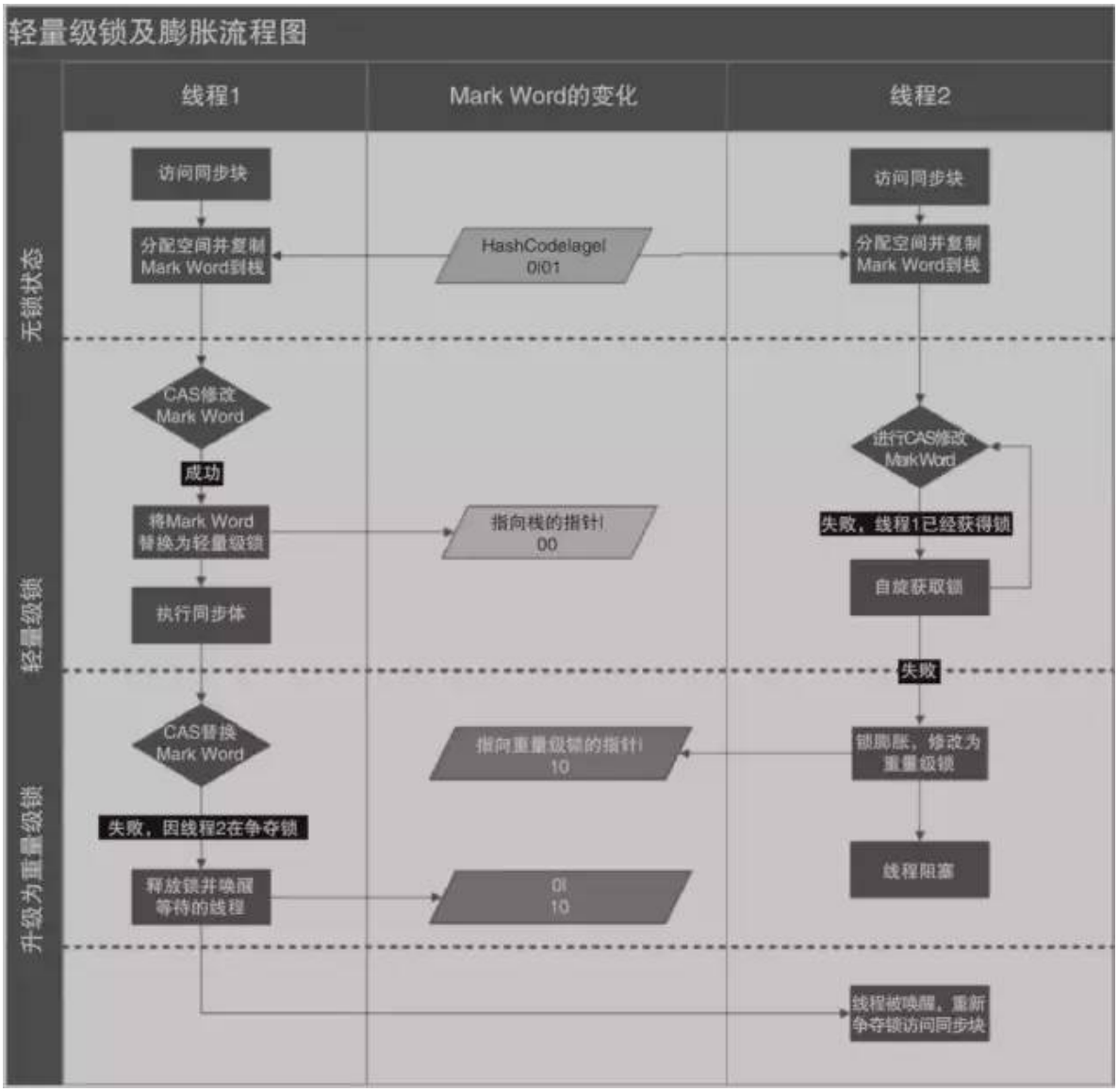
JavaSE1.6为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”。

在JavaSE1.6中，锁一共有4种状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁状态和重量级锁状态，这几个状态会随着竞争情况逐渐升级。

锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率。

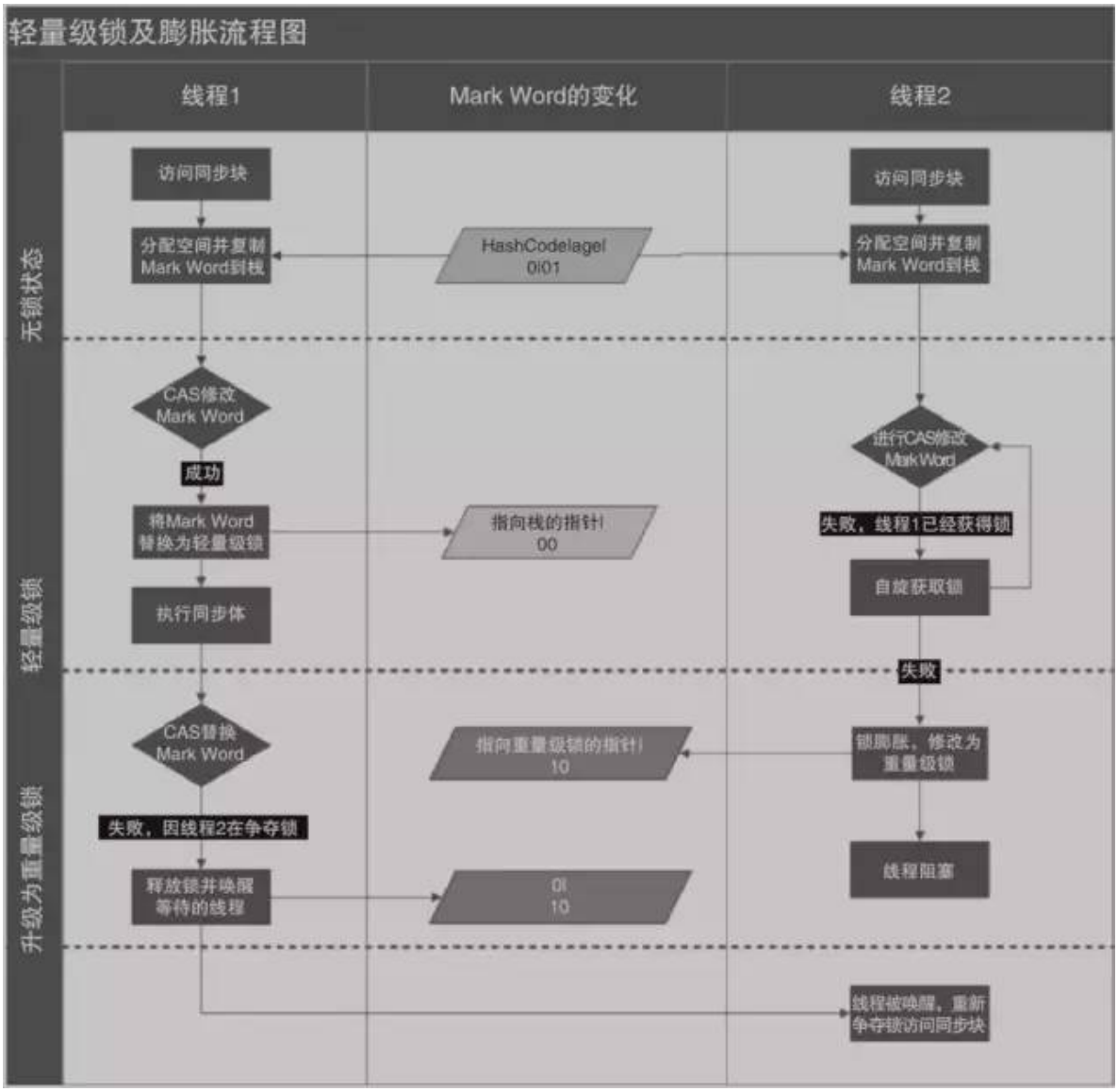
偏向锁：

> 无锁竞争的情况下为了减少锁竞争的资源开销，引入偏向锁。



轻量级锁：

> 轻量级锁所适应的场景是线程交替执行同步块的情况。



锁粗化（Lock Coarsening）： 也就是减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁。

锁消除（Lock Elimination）： 锁削除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行削除。

适应性自旋（Adaptive Spinning）： 自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。另一方面，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。

2.2.4 锁的优缺点对比

锁	优　点	缺　点	适用场景
偏向锁	加锁和解锁不需要额外的消耗，和执行非同步方法相比仅存在纳秒级的差距	如果线程间存在锁竞争，会带来额外的锁撤销的消耗	适用于只有一个线程访问同步块场景
轻量级锁	竞争的线程不会阻塞，提高了程序的响应速度	如果始终得不到锁竞争的线程，使用自旋会消耗 CPU	追求响应时间 同步块执行速度非常快
重量级锁	线程竞争不使用自旋，不会消耗 CPU	线程阻塞，响应时间缓慢	追求吞吐量 同步块执行速度较长

2.3、CAS

> CAS，在Java并发应用中通常指CompareAndSwap或CompareAndSet，即比较并交换。

1、CAS是一个原子操作，它比较一个内存位置的值并且只有相等时修改这个内存位置的值为新的值，保证了新的值总是基于最新的信息计算的，如果有其他线程在这期间修改了这个值则CAS失败。CAS返回是否成功或者内存位置原来的值用于判断是否CAS成功。

2、JVM中的CAS操作是利用了处理器提供的CMPXCHG指令实现的。

优点：

- 竞争不大的时候系统开销小。

缺点：

- 循环时间长开销大。
- ABA问题。
- 只能保证一个共享变量的原子操作。

3、Java中的锁实现

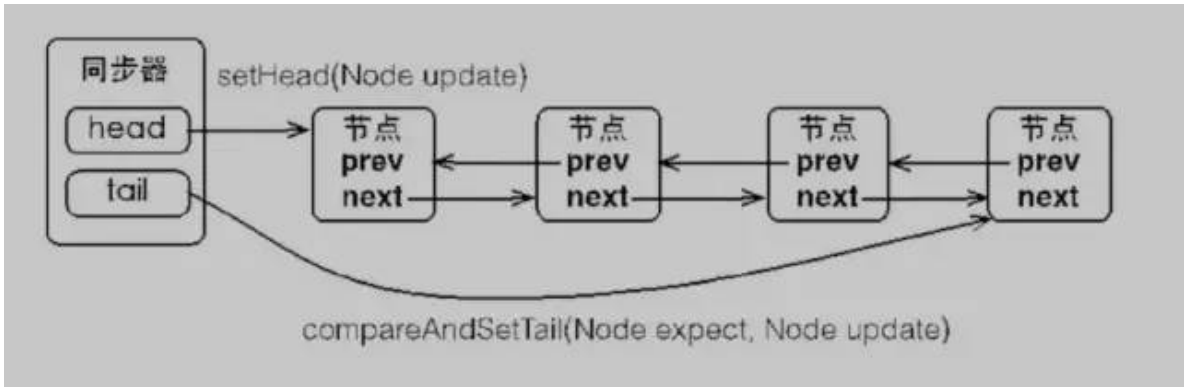
3.1、队列同步器（AQS）

> 队列同步器AbstractQueuedSynchronizer（以下简称同步器），是用来构建锁或者其他同步组件的基础框架。

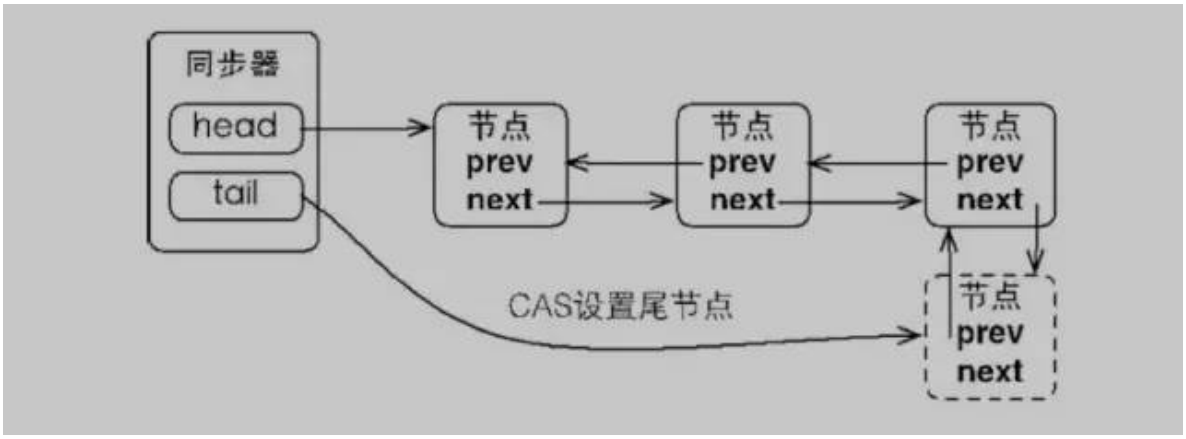
3.1.1、它使用了一个int成员变量表示同步状态。

3.1.2、通过内置的FIFO双向队列来完成获取锁线程的排队工作。

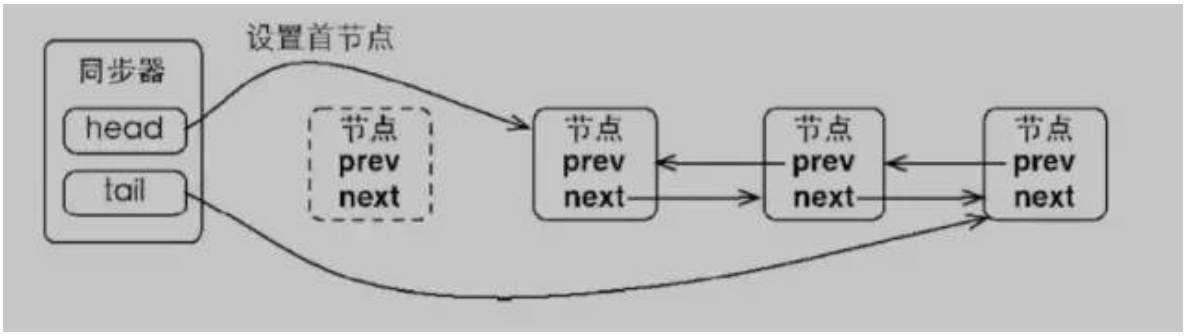
- 同步器包含两个节点类型的应用，一个指向头节点，一个指向尾节点，未获取到锁的线程会创建节点线程安全（compareAndSetTail）的加入队列尾部。同步队列遵循FIFO，首节点是获取同步状态成功的节点。



- 未获取到锁的线程将创建一个节点，设置到尾节点。如下图所示：



- 首节点的线程在释放锁时，将会唤醒后继节点。而后继节点将会在获取锁成功时将自己设置为首节点。如下图所示：

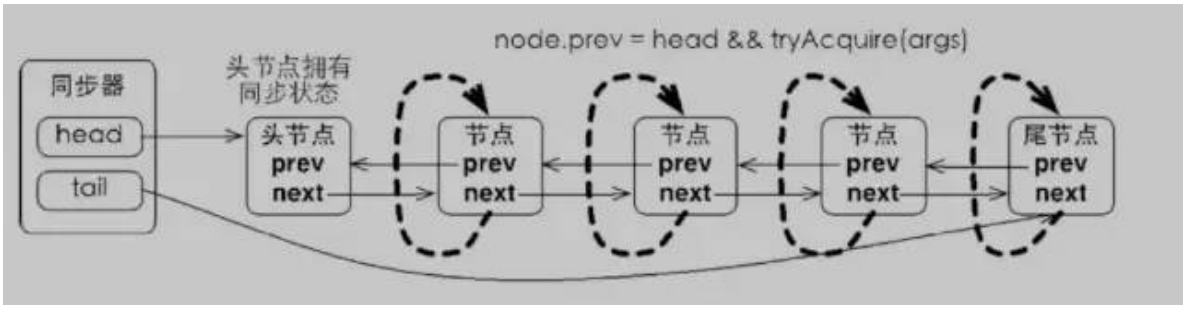


3.1.3、独占式/共享式锁获取

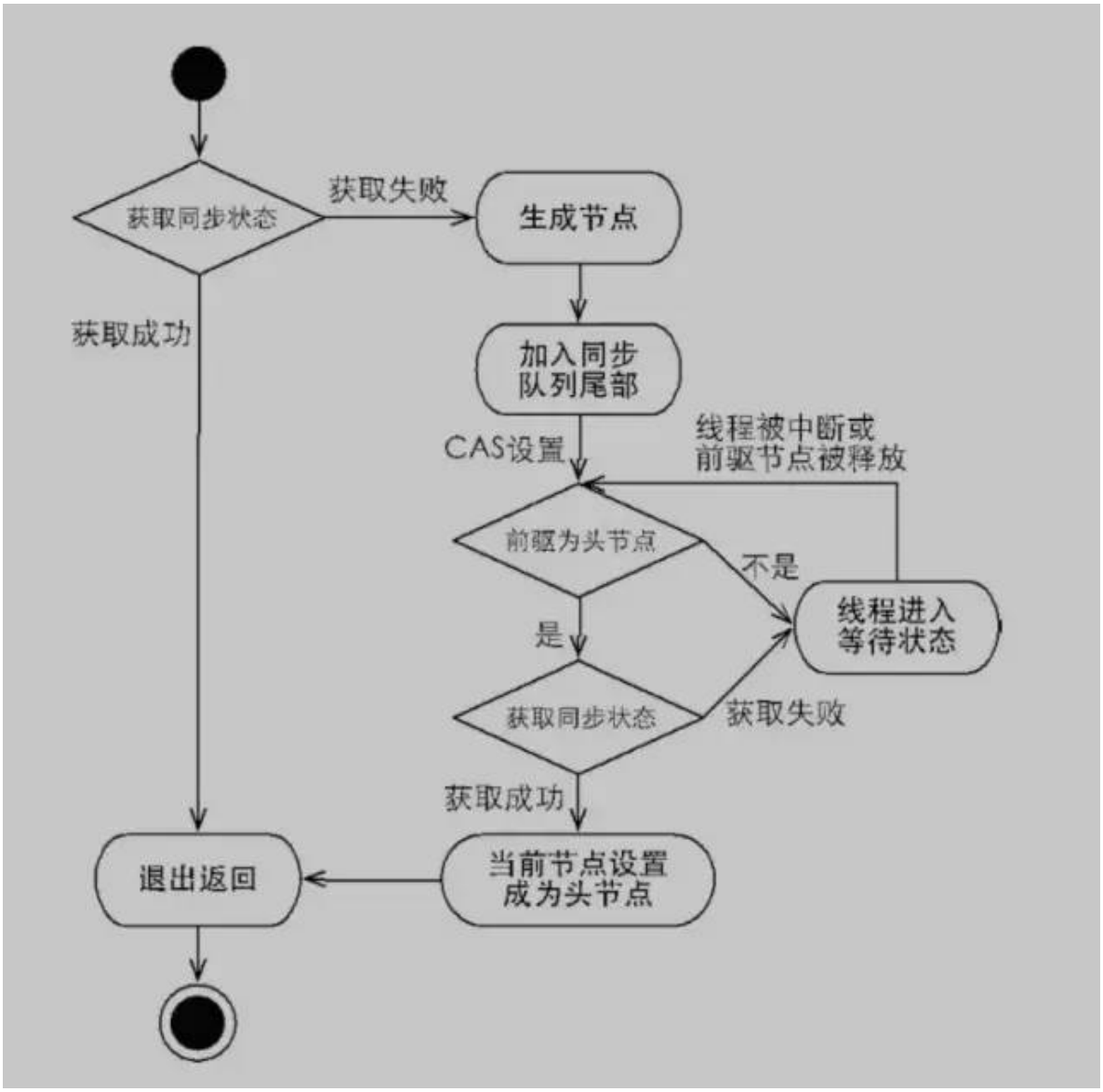
> 独占式：有且只有一个线程能获取到锁，如：ReentrantLock。
>
> 共享式：可以多个线程同时获取到锁，如：CountDownLatch

独占式

- 每个节点自旋观察自己的前一节点是不是Header节点，如果是，就去尝试获取锁。

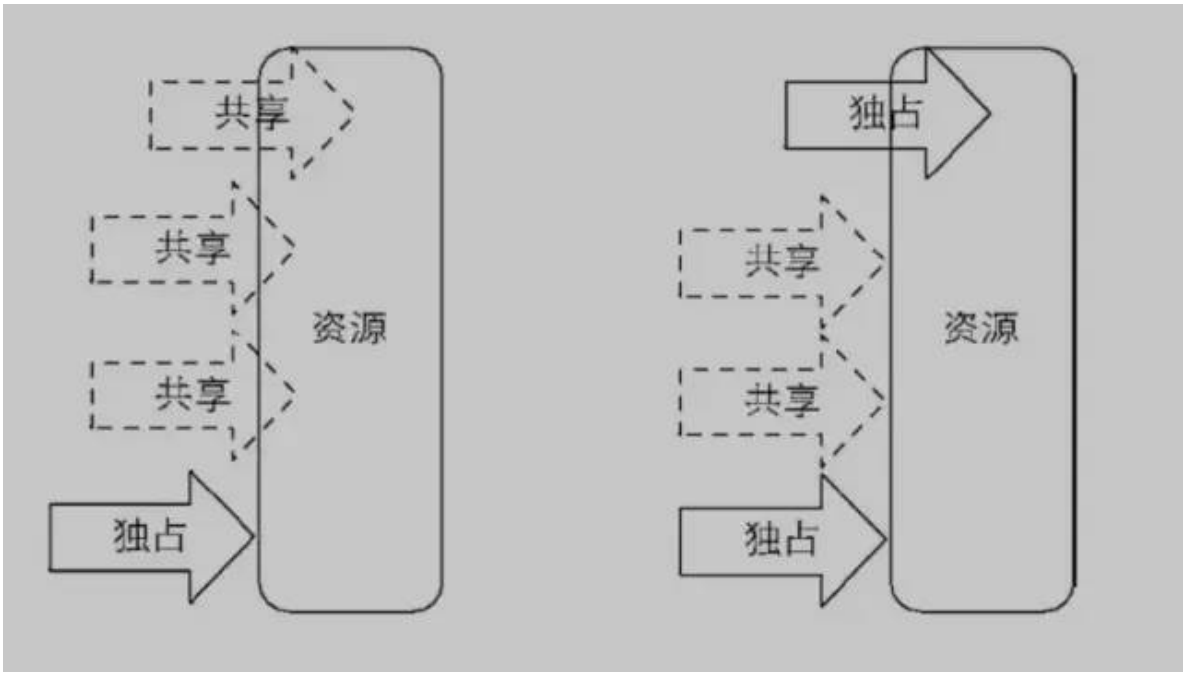


- 独占式锁获取流程：

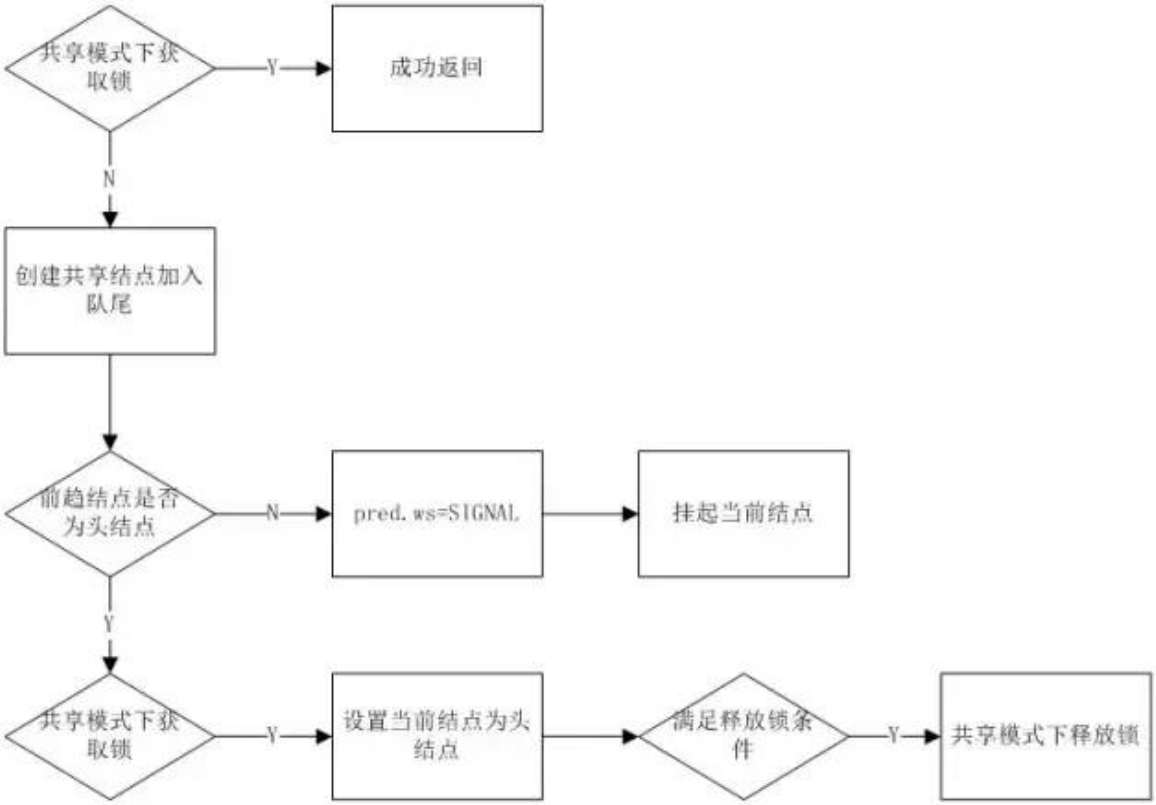


共享式：

- 共享式与独占式的区别：

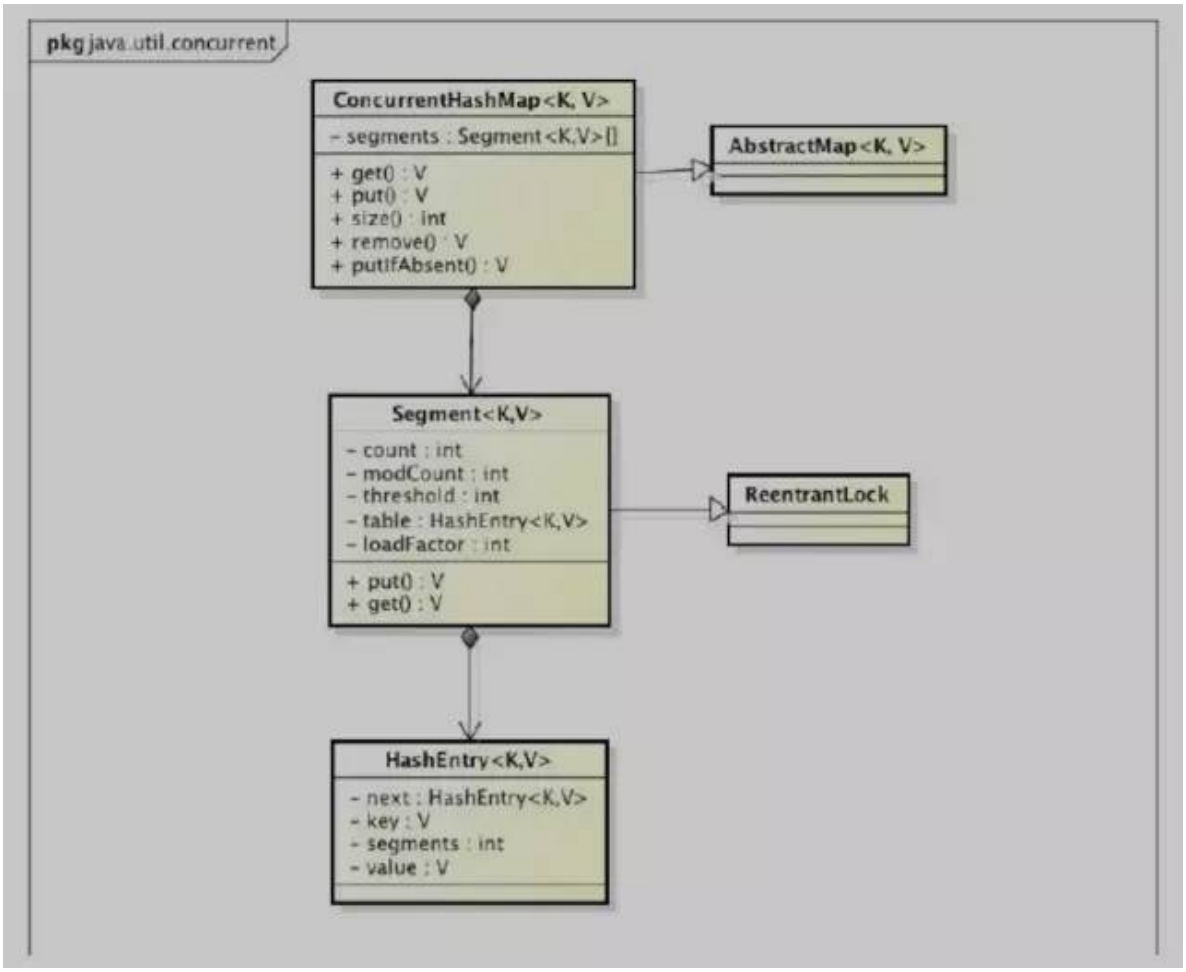


- 共享锁获取流程：

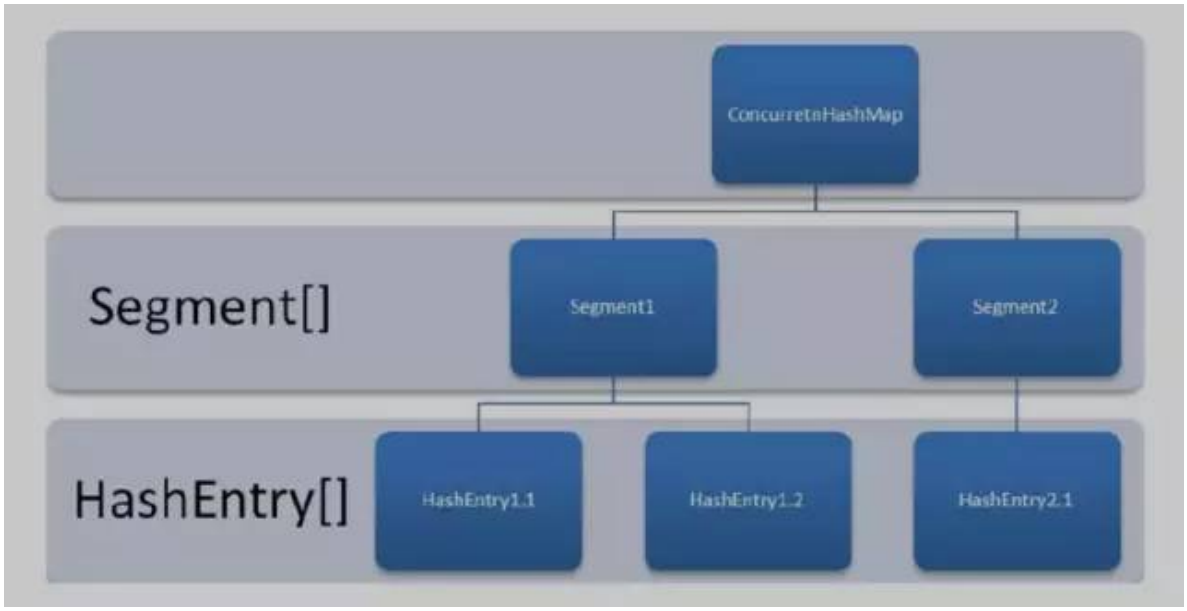


4、锁的使用

4.1、ConcurrentHashMap的实现原理及使用（1.7）



ConcurrentHashMap类图



ConcurrentHashMap数据结构

结论：ConcurrentHashMap使用的锁分段技术。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

显示锁

Java提供一系列的显示锁类，均位于java.util.concurrent.locks包中。

锁的分类： 排他锁，共享锁

- 排他锁又被称为独占锁，即读写互斥、写写互斥、读读互斥。
- Java的ReadWriteLock是一种共享锁，提供读读共享，但读写和写写仍然互斥。

可重入锁

1. ReentrantLock：调用ReentrantLock对象的lock()方法获取锁,调用unlock()方法释放锁。
   （ ReentrantLockTest ）
2. 使用ReentrantLock实现同步。(ConditionTestMoreMethod)
3. 公平与非公平锁: new ReentrantLock(isFair) 默认采用非公平锁 公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出顺序。
   非公平锁就是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一-定先得到锁，这个方式可能造成某些线程一 直拿不到锁，结果也就是不公平的了。(Fair_noFair_test)
4. ReentrantLock排它锁，线程安全但是效率低下。(lockMethods)

读写锁

ReadWriteLock接口、ReentrantReadWriteLock

- ReadWriteLock接口：同一时刻允许多个读线程同时访问，但是写线程访问的时候，所有的读和写都被阻塞。
- ReentrantReadWriteLock实现了ReadWriteLock接口。

ReentrantReadWriteLock 读写锁表示也有两个锁，一个是读操作相关的锁，也称为共享锁;另一个是写操作相关的锁，也叫排他锁。也就是多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥。在没有线程Thread进行写入操作时，进行读取操作的多个Thread都可以获取读锁，而进行写人操作的Thread只有在获取写锁后才能进行写人操作。即多个Thread可以同时进行读取操作，但是同一时刻只允许一个Thread进行写人操作。

- 读读不互斥。(ReadWriteLockBegin1)
- 写写互斥。(ReadWriteLockBegin2)
- 读写互斥。(ReadWriteLockBegin3)
- 写读互斥。(ReadWriteLockBegin4)

乐观锁

悲观锁

自旋锁

AQS详解

一、概述

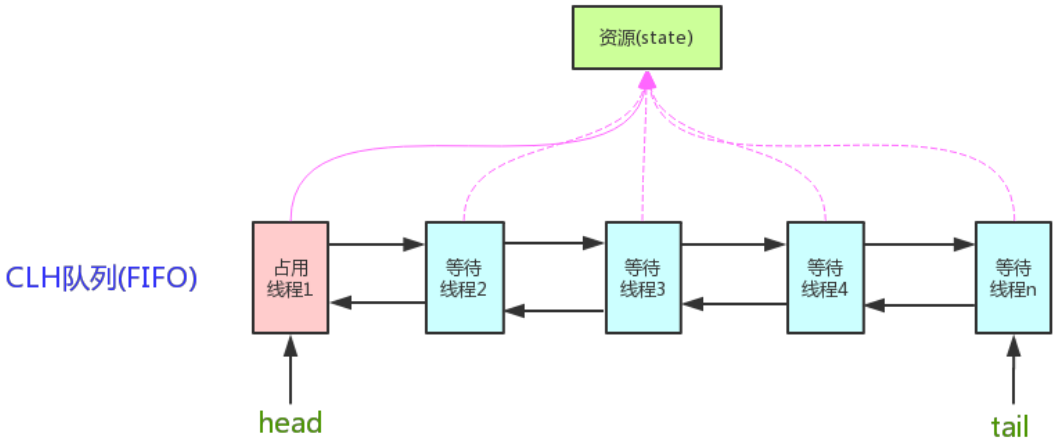
谈到并发，不得不谈ReentrantLock；而谈到ReentrantLock，不得不AbstractQueuedSynchronizer（AQS）！

类如其名，抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch...。

以下是本文的目录大纲：

1. 概述
2. 框架
3. 源码详解
4. 简单应用

二、框架



它维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里volatile是核心关键词，具体volatile的语义，在此不述。state的访问方式有三种:

- getState()
- setState()
- compareAndSetState()

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

三、源码详解

本节开始讲解AQS的源码实现。依照acquire-release、acquireShared-releaseShared的次序来。

3.1 acquire(int)

此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了。下面是acquire()的源码：

```
public final void acquire(int arg) {
     if (!tryAcquire(arg) &&
         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
         selfInterrupt();
 }
```

函数流程如下：

1. tryAcquire()尝试直接去获取资源，如果成功则直接返回；
2. addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

这时单凭这4个抽象的函数来看流程还有点朦胧，不要紧，看完接下来的分析后，你就会明白了。就像《大话西游》里唐僧说的：等你明白了舍生取义的道理，你自然会回来和我唱这首歌的。

3.1.1 tryAcquire(int)

此方法尝试去获取独占资源。如果获取成功，则直接返回true，否则直接返回false。这也正是tryLock()的语义，还是那句话，当然不仅仅只限于tryLock()。如下是tryAcquire()的源码：

```
protected boolean tryAcquire(int arg) {
        throw new UnsupportedOperationException();
    }
```

什么？直接throw异常？说好的功能呢？好吧，**还记得概述里讲的AQS只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现吗**？**就是这里了！！！AQS这里只定义了一个接口，具体资源的获取交由自定义同步器去实现了（通过state的get/set/CAS）！！！至于能不能重入，能不能加塞，那就看具体的自定义同步器怎么去设计了！！！当然，自定义同步器在进行资源访问时要考虑线程安全的影响。

这里之所以没有定义成abstract，是因为独占模式下只用实现tryAcquire-tryRelease，而共享模式下只用实现tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口。说到底，Doug Lea还是站在咱们开发者的角度，尽量减少不必要的工作量。

3.1.2 addWaiter(Node)

此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点。还是上源码吧：

```
private Node addWaiter(Node mode) {
    //以给定模式构造结点。mode有两种：EXCLUSIVE（独占）和SHARED（共享）
    Node node = new Node(Thread.currentThread(), mode);

    //尝试快速方式直接放到队尾。
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }

    //上一步失败则通过enq入队。
    enq(node);
    return node;
}
```

不用再说了，直接看注释吧。这里我们说下Node。Node结点是对每一个访问同步代码的线程的封装，其包含了需要同步的线程本身以及线程的状态，如是否被阻塞，是否等待唤醒，是否已经被取消等。变量waitStatus则表示当前被封装成Node结点的等待状态，共有4种取值CANCELLED、SIGNAL、CONDITION、PROPAGATE。

- CANCELLED：值为1，在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消该Node的结点，其结点的waitStatus为CANCELLED，即结束状态，进入该状态后的结点将不会再变化。
- SIGNAL：值为-1，被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。说白了，就是处于唤醒状态，只要前继结点释放锁，就会通知标识为SIGNAL状态的后继结点的线程执行。
- CONDITION：值为-2，与Condition相关，该标识的结点处于**等待队列中**，结点的线程等待在Condition上，当其他线程调用了Condition的signal()方法后，CONDITION状态的结点将**从等待队列转移到同步队列中**，等待获取同步锁。

- PROPAGATE：值为-3，与共享模式相关，在共享模式中，该状态标识结点的线程处于可运行状态。
- 0状态：值为0，代表初始化状态。

AQS在判断状态时，通过用waitStatus>0表示取消状态，而waitStatus<0表示有效状态。

3.1.2.1 enq(Node)

此方法用于将node加入队尾。源码如下：

```
private Node enq(final Node node) {
    //CAS"自旋"，直到成功加入队尾
    for (;;) {
        Node t = tail;
        if (t == null) { // 队列为空，创建一个空的标志结点作为head结点，并将tail也指向它。
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {//正常流程，放入队尾
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
```

如果你看过AtomicInteger.getAndIncrement()函数源码，那么相信你一眼便看出这段代码的精华。CAS自旋volatile变量，是一种很经典的用法。还不太了解的，自己去百度一下吧。

3.1.3 acquireQueued(Node, int)

OK，通过tryAcquire()和addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了。聪明的你立刻应该能想到该线程下一部该干什么了吧：**进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了**。没错，就是这样！是不是跟医院排队拿号有点相似~~acquireQueued()就是干这件事：**在等待队列中排队拿号（中间没其它事干可以休息），直到拿到号后再返回。**这个函数非常关键，还是上源码吧：

```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;//标记是否成功拿到资源
    try {
        boolean interrupted = false;//标记等待过程中是否被中断过

        //又是一个“自旋”！
        for (;;) {
            final Node p = node.predecessor();//拿到前驱
            //如果前驱是head，即该结点已成老二，那么便有资格去尝试获取资源（可能是老大释放完资源唤醒自己的，当然也可能被interrupt了）。
            if (p == head && tryAcquire(arg)) {
                setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。
                p.next = null; // setHead中node.prev已置为null，此处再将head.next置为null，就是为了方便GC回收以前的head结点。也就意味着之前拿完资源的结点出队了！
                failed = false;
                return interrupted;//返回等待过程中是否被中断过
            }
```

```
            //如果自己可以休息了，就进入waiting状态，直到被unpark()
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;//如果等待过程中被中断过，哪怕只有那么一次，就将
interrupted标记为true
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

到这里了，我们先不急着总结acquireQueued()的函数流程，先看看shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()具体干些什么。

3.1.3.1 shouldParkAfterFailedAcquire(Node, Node)

此方法主要用于检查状态，看看自己是否真的可以去休息了，万一队列前边的线程都放弃了只是瞎站着，那也说不定，对吧！

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;//拿到前驱的状态
    if (ws == Node.SIGNAL)
        //如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了
        return true;
    if (ws > 0) {
        /*
         * 如果前驱放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。
         * 注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，稍
后就会被保安大叔赶走了(GC回收)！
         */
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
         //如果前驱正常，那就把前驱的状态设置成SIGNAL，告诉它拿完号后通知自己一下。有可能失
败，人家说不定刚刚释放完呢！
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心去休息，需要去找个安心的休息点，同时可以再尝试下看有没有机会轮到自己拿号。

3.1.3.2 parkAndCheckInterrupt()

如果线程找好安全休息点后，那就可以安心去休息了。此方法就是让线程去休息，真正进入等待状态。

```
private final boolean parkAndCheckInterrupt() {
     LockSupport.park(this);//调用park()使线程进入waiting状态
     return Thread.interrupted();//如果被唤醒，查看自己是不是被中断的。
 }
```

park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt()。（再说一句，如果线程状态转换不熟，可以参考本人写的Thread详解）。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

3.1.3.3 小结

OK，看了shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()，现在让我们再回到acquireQueued()，总结下该函数的具体流程：

1. 结点进入队尾后，检查状态，找到安全休息点；
2. 调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；
3. 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

3.1.4 小结

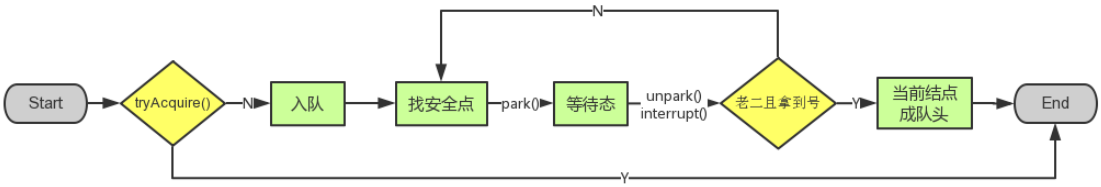
OKOK，acquireQueued()分析完之后，我们接下来再回到acquire()！再贴上它的源码吧：

```
public final void acquire(int arg) {
     if (!tryAcquire(arg) &&
         acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
         selfInterrupt();
 }
```

再来总结下它的流程吧：

1. 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回；
2. 没成功，则addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

由于此函数是重中之重，我再用流程图总结一下：



至此，acquire()的流程终于算是告一段落了。这也就是ReentrantLock.lock()的流程，不信你去看其lock()源码吧，整个函数就是一条acquire(1)！！！

3.2 release(int)

上一小节已经把acquire()说完了，这一小节就来讲讲它的反操作release()吧。此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;//找到头结点
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);//唤醒等待队列里的下一个线程
        return true;
    }
    return false;
}
```

逻辑并不复杂。它调用tryRelease()来释放资源。有一点需要注意的是，**它是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！**所以自定义同步器在设计tryRelease()的时候要明确这一点！！**

3.2.1 tryRelease(int)

此方法尝试去释放指定量的资源。下面是tryRelease()的源码：

```
protected boolean tryRelease(int arg) {
     throw new UnsupportedOperationException();
 }
```

跟tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，上面已经提到了，release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false。

3.2.2 unparkSuccessor(Node)

此方法用于唤醒等待队列中下一个线程。下面是源码：

```
private void unparkSuccessor(Node node) {
    //这里，node一般为当前线程所在的结点。
    int ws = node.waitStatus;
    if (ws < 0)//置零当前线程所在的结点状态，允许失败。
        compareAndSetWaitStatus(node, ws, 0);

    Node s = node.next;//找到下一个需要唤醒的结点s
    if (s == null || s.waitStatus > 0) {//如果为空或已取消
        s = null;
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)//从这里可以看出，<=0的结点，都是还有效的结点。
                s = t;
    }
    if (s != null)
        LockSupport.unpark(s.thread);//唤醒
}
```

这个函数并不复杂。一句话概括：**用unpark()唤醒等待队列中最前边的那个未放弃线程**，这里我们也用s来表示吧。此时，再和acquireQueued()联系起来，s被唤醒后，进入if (p == head && tryAcquire(arg))的判断（即使p!=head也没关系，它会再进入shouldParkAfterFailedAcquire()寻找一个安全点。这里既然s已经是等待队列中最前边的那个未放弃线程了，那么通过shouldParkAfterFailedAcquire()的调整，s也必然会跑到head的next结点，下一次自旋p==head就成

立啦），然后s把自己设置成head标杆结点，表示自己已经获取到资源了，acquire()也返回了！！And then, DO what you WANT!

3.2.3 小结

release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

3.3 acquireShared(int)

此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。下面是acquireShared()的源码：

```
public final void acquireShared(int arg) {
     if (tryAcquireShared(arg) < 0)
         doAcquireShared(arg);
 }
```

这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取。所以这里acquireShared()的流程就是：

1. tryAcquireShared()尝试获取资源，成功则直接返回；
2. 失败则通过doAcquireShared()进入等待队列，直到获取到资源为止才返回。

3.3.1 doAcquireShared(int)

此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。下面是doAcquireShared()的源码：

```
private void doAcquireShared(int arg) {
    final Node node = addWaiter(Node.SHARED);//加入队列尾部
    boolean failed = true;//是否成功标志
    try {
        boolean interrupted = false;//等待过程中是否被中断过的标志
        for (;;) {
            final Node p = node.predecessor();//前驱
            if (p == head) {//如果到head的下一个，因为head是拿到资源的线程，此时node被唤醒，很可能是head用完资源来唤醒自己的
                int r = tryAcquireShared(arg);//尝试获取资源
                if (r >= 0) {//成功
                    setHeadAndPropagate(node, r);//将head指向自己，还有剩余资源可以再唤醒之后的线程
                    p.next = null; // help GC
                    if (interrupted)//如果等待过程中被打断过，此时将中断补上。
                        selfInterrupt();
                    failed = false;
                    return;
                }
            }

            //判断状态，寻找安全点，进入waiting状态，等着被unpark()或interrupt()
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
```

```
            cancelAcquire(node);
    }
}
```

有木有觉得跟acquireQueued()很相似？对，其实流程并没有太大区别。只不过这里将补中断的selfInterrupt()放到doAcquireShared()里了，而独占模式是放到acquireQueued()之外，其实都一样，不知道Doug Lea是怎么想的。

跟独占模式比，还有一点需要注意的是，这里只有线程是head.next时（“老二”），才会去尝试获取资源，有剩余的话还会唤醒之后的队友。那么问题就来了，假如老大用完后释放了5个资源，而老二需要6个，老三需要1个，老四需要2个。老大先唤醒老二，老二一看资源不够，他是把资源让给老三呢，还是不让？答案是否定的！老二会继续park()等待其他线程释放资源，也更不会去唤醒老三和老四了。独占模式，同一时刻只有一个线程去执行，这样做未尝不可；但共享模式下，多个线程是可以同时执行的，现在因为老二的资源需求量大，而把后面量小的老三和老四也都卡住了。当然，这并不是问题，只是AQS保证严格按照入队顺序唤醒罢了（保证公平，但降低了并发）。

3.3.1.1 setHeadAndPropagate(Node, int)

```
private void setHeadAndPropagate(Node node, int propagate) {
    Node h = head;
    setHead(node);//head指向自己
     //如果还有剩余量，继续唤醒下一个邻居线程
    if (propagate > 0 || h == null || h.waitStatus < 0) {
        Node s = node.next;
        if (s == null || s.isShared())
            doReleaseShared();
    }
}
```

此方法在setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合（比如还有剩余资源），还会去唤醒后继结点，毕竟是共享模式！

doReleaseShared()我们留着下一小节的releaseShared()里来讲。

3.3.2 小结

OK，至此，acquireShared()也要告一段落了。让我们再梳理一下它的流程：

1. tryAcquireShared()尝试获取资源，成功则直接返回；
2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

其实跟acquire()的流程大同小异，只不过多了个**自己拿到资源后，还会去唤醒后继队友的操作（这才是共享嘛）。**

3.4 releaseShared()

上一小节已经把acquireShared()说完了，这一小节就来讲讲它的反操作releaseShared()吧。此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果成功释放且允许唤醒等待线程，它会唤醒等待队列里的其他线程来获取资源。下面是releaseShared()的源码：

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {//尝试释放资源
        doReleaseShared();//唤醒后继结点
        return true;
    }
    return false;
}
```

此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于独占下可重入的考量；而共享模式下的releaseShared()则没有这种要求，共享模式实质就是控制一定量的线程并发执行，那么拥有资源的线程在释放掉部分资源时就可以唤醒后继等待结点。例如，资源总量是13，A（5）和B（7）分别获取到资源并发运行，C（4）来时只剩1个资源就需要等待。A在运行过程中释放掉2个资源量，然后tryReleaseShared(2)返回true唤醒C，C一看只有3个仍不够继续等待；随后B又释放2个，tryReleaseShared(2)返回true唤醒C，C一看有5个够自己用了，然后C就可以跟A和B一起运行。而ReentrantReadWriteLock读锁的tryReleaseShared()只有在完全释放掉资源（state=0）才返回true，所以自定义同步器可以根据需要决定tryReleaseShared()的返回值。

3.4.1 doReleaseShared()

此方法主要用于唤醒后继。下面是它的源码：

```
private void doReleaseShared() {
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {
                if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                    continue;
                unparkSuccessor(h);//唤醒后继
            }
            else if (ws == 0 &&
                     !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                continue;
        }
        if (h == head)// head发生变化
            break;
    }
}
```

3.5 小结

本节我们详解了独占和共享两种模式下获取-释放资源(acquire-release、acquireShared-releaseShared)的源码，相信大家都有一定认识了。值得注意的是，acquire()和acquireShared()两种方法下，线程在等待队列中都是忽略中断的。AQS也支持响应中断的，acquireInterruptibly()/acquireSharedInterruptibly()即是，这里相应的源码跟acquire()和acquireShared()差不多，这里就不再详解了。

四、简单应用

通过前边几个章节的学习，相信大家已经基本理解AQS的原理了。这里再将“框架”一节中的一段话复制过来：

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

OK，下面我们就以AQS源码里的Mutex为例，讲一下AQS的简单应用。

4.1 Mutex（互斥锁）

Mutex是一个不可重入的互斥锁实现。锁资源（AQS里的state）只有两种状态：0表示未锁定，1表示锁定。下边是Mutex的核心源码：

```
class Mutex implements Lock, java.io.Serializable {
    // 自定义同步器
    private static class Sync extends AbstractQueuedSynchronizer {
        // 判断是否锁定状态
        protected boolean isHeldExclusively() {
            return getState() == 1;
        }

        // 尝试获取资源，立即返回。成功则返回true，否则false。
        public boolean tryAcquire(int acquires) {
            assert acquires == 1; // 这里限定只能为1个量
            if (compareAndSetState(0, 1)) {//state为0才设置为1，不可重入！
                setExclusiveOwnerThread(Thread.currentThread());//设置为当前线程独
占资源
                return true;
            }
            return false;
        }

        // 尝试释放资源，立即返回。成功则为true，否则false。
        protected boolean tryRelease(int releases) {
            assert releases == 1; // 限定为1个量
            if (getState() == 0)//既然来释放，那肯定就是已占有状态了。只是为了保险，多层
判断！
                throw new IllegalMonitorStateException();
            setExclusiveOwnerThread(null);
            setState(0);//释放资源，放弃占有状态
            return true;
        }
    }

    // 真正同步类的实现都依赖继承于AQS的自定义同步器！
    private final Sync sync = new Sync();

    //lock<-->acquire。两者语义一样：获取资源，即便等待，直到成功才返回。
    public void lock() {
        sync.acquire(1);
    }
```

```
    //tryLock<-->tryAcquire。两者语义一样：尝试获取资源，要求立即返回。成功则为true，失败则为false。
    public boolean tryLock() {
        return sync.tryAcquire(1);
    }

    //unlock<-->release。两者语文一样：释放资源。
    public void unlock() {
        sync.release(1);
    }

    //锁是否占有状态
    public boolean isLocked() {
        return sync.isHeldExclusively();
    }
}
```

同步类在实现时一般都将自定义同步器（sync）定义为内部类，供自己使用；而同步类自己（Mutex）则实现某个接口，对外服务。当然，接口的实现要直接依赖sync，它们在语义上也存在某种对应关系！！而sync只用实现资源state的获取-释放方式tryAcquire-tryRelelase，至于线程的排队、等待、唤醒等，上层的AQS都已经实现好了，我们不用关心。

除了Mutex，ReentrantLock/CountDownLatch/Semphore这些同步类的实现方式都差不多，不同的地方就在获取-释放资源的方式tryAcquire-tryRelelase。掌握了这点，AQS的核心便被攻破了！

五、AQS之CLH同步队列

AQS内部维护着一个FIFO队列，该队列就是CLH同步队列。

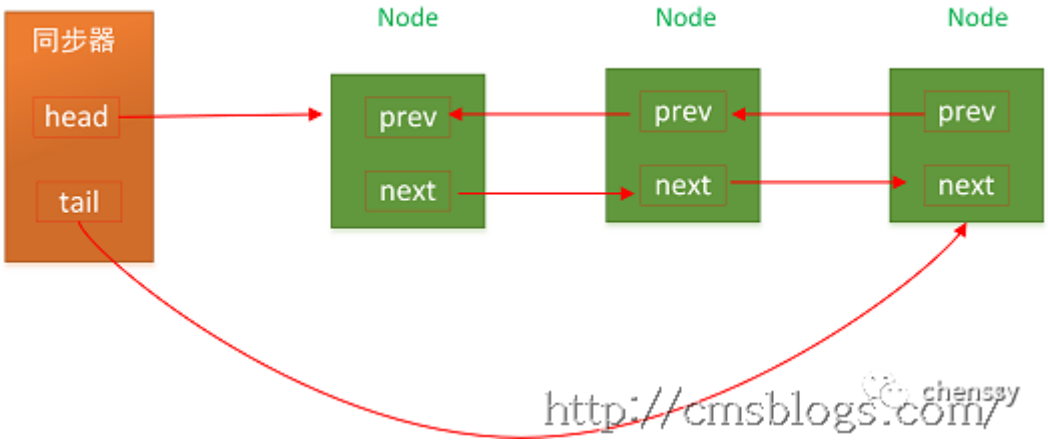
CLH同步队列是一个FIFO双向队列，AQS依赖它来完成同步状态的管理，当前线程如果获取同步状态失败时，AQS则会将当前线程已经等待状态等信息构造成一个节点（Node）并将其加入到CLH同步队列，同时会阻塞当前线程，当同步状态释放时，会把首节点唤醒（公平锁），使其再次尝试获取同步状态。

在CLH同步队列中，一个节点表示一个线程，它保存着线程的引用（thread）、状态（waitStatus）、前驱节点（prev）、后继节点（next），其定义如下：

```
static final class Node {
    /** 共享 */
    static final Node SHARED = new Node();

    /** 独占 */
    static final Node EXCLUSIVE = null;

    /**
     * 因为超时或者中断，节点会被设置为取消状态，被取消的节点时不会参与到竞争中的，他会一直保持取消状态不会转变为其他状态；
     */
    static final int CANCELLED =  1;

    /**
     * 后继节点的线程处于等待状态，而当前节点的线程如果释放了同步状态或者被取消，将会通知后继节点，使后继节点的线程得以运行
     */
    static final int SIGNAL    = -1;
```

```
    /**
     * 节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal()
后，改节点将会从等待队列中转移到同步队列中，加入到同步状态的获取中
     */
    static final int CONDITION = -2;

    /**
     * 表示下一次共享式同步状态获取将会无条件地传播下去
     */
    static final int PROPAGATE = -3;

    /** 等待状态 */
    volatile int waitStatus;

    /** 前驱节点 */
    volatile Node prev;

    /** 后继节点 */
    volatile Node next;

    /** 获取同步状态的线程 */
    volatile Thread thread;

    Node nextWaiter;

    final boolean isShared() {
        return nextWaiter == SHARED;
    }

    final Node predecessor() throws NullPointerException {
        Node p = prev;
        if (p == null)
            throw new NullPointerException();
        else
            return p;
    }

    Node() {
    }

    Node(Thread thread, Node mode) {
        this.nextWaiter = mode;
        this.thread = thread;
    }

    Node(Thread thread, int waitStatus) {
        this.waitStatus = waitStatus;
        this.thread = thread;
    }}
```

CLH同步队列结构图如下：



入列

学了数据结构的我们，CLH队列入列是再简单不过了，无非就是tail指向新节点、新节点的prev指向当前最后的节点，当前最后一个节点的next指向当前节点。

代码我们可以看看addWaiter(Node node)方法：

```
    private Node addWaiter(Node mode) {
        //新建Node
        Node node = new Node(Thread.currentThread(), mode);
        //快速尝试添加尾节点
        Node pred = tail;
        if (pred != null) {
            node.prev = pred;
            //CAS设置尾节点
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node;
            }
        }
        //多次尝试
        enq(node);
        return node;
    }
```

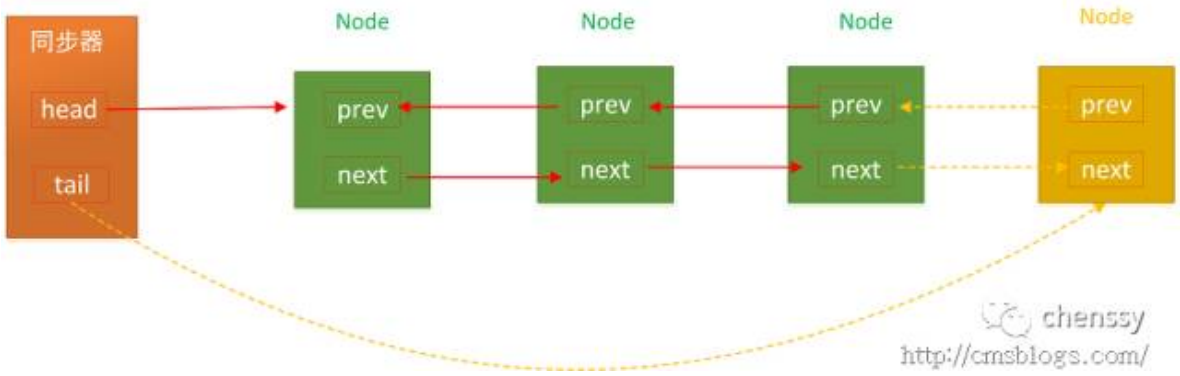
addWaiter(Node node)先通过快速尝试设置尾节点，如果失败，则调用enq(Node node)方法设置尾节点。

```
    private Node enq(final Node node) {
        //多次尝试，直到成功为止
        for (;;) {
            Node t = tail;
            //tail不存在，设置为首节点
            if (t == null) {
                if (compareAndSetHead(new Node()))
                    tail = head;
            } else {
                //设置为尾节点
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;
                }
```

```
            }
        }
    }
```

在上面代码中，两个方法都是通过一个CAS方法compareAndSetTail(Node expect, Node update)来设置尾节点，该方法可以确保节点是线程安全添加的。在enq(Node node)方法中，AQS通过“死循环”的方式来保证节点可以正确添加，只有成功添加后，当前线程才会从该方法返回，否则会一直执行下去。

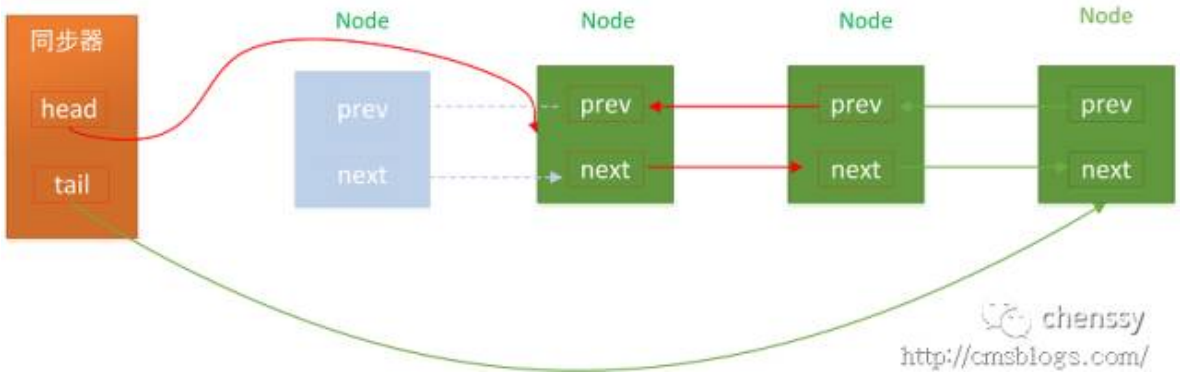
过程图如下：



出列

CLH同步队列遵循FIFO，首节点的线程释放同步状态后，将会唤醒它的后继节点（next），而后继节点将会在获取同步状态成功时将自己设置为首节点，这个过程非常简单，head执行该节点并断开原首节点的next和当前节点的prev即可，注意在这个过程是不需要使用CAS来保证的，因为只有一个线程能够成功获取到同步状态。

过程图如下：



并发容器深度剖析

Java并发容器：ConcurrentHashMap

HashMap是我们用得非常频繁的一个集合，但是由于它是非线程安全的，在多线程环境下，put操作是有可能产生死循环的，导致CPU利用率接近100%。为了解决该问题，提供了Hashtable和Collections.synchronizedMap(hashMap)两种解决方案，但是这两种方案都是对读写加锁，独占式，一个线程在读时其他线程必须等待，吞吐量较低，性能较为低下。故而Doug Lea大神给我们提供了高性能的线程安全HashMap：ConcurrentHashMap。

ConcurrentHashMap的实现

ConcurrentHashMap作为Concurrent一族，其有着高效地并发操作，相比Hashtable的笨重，ConcurrentHashMap则更胜一筹了。 在1.8版本以前，ConcurrentHashMap采用分段锁的概念，使锁更加细化，但是1.8已经改变了这种思路，而是利用CAS+Synchronized来保证并发更新的安全，当然底层采用数组+链表+红黑树的存储结构。 关于1.7和1.8的区别请参考占小狼博客：谈谈ConcurrentHashMap1.7和1.8的不同实现:http://www.jianshu.com/p/e694f1e868ec 我们从如下几个部分全面了解ConcurrentHashMap在1.8中是如何实现的：

1. 重要概念
2. 重要内部类
3. ConcurrentHashMap的初始化
4. put操作
5. get操作
6. size操作
7. 扩容
8. 红黑树转换

重要概念

ConcurrentHashMap定义了如下几个常量：

```
// 最大容量：2^30=1073741824
private static final int MAXIMUM_CAPACITY = 1 << 30;

// 默认初始值，必须是2的幕数
private static final int DEFAULT_CAPACITY = 16;

//
static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

//
private static final int DEFAULT_CONCURRENCY_LEVEL = 16;

//
private static final float LOAD_FACTOR = 0.75f;

// 链表转红黑树阀值,> 8 链表转换为红黑树
static final int TREEIFY_THRESHOLD = 8;

//树转链表阀值，小于等于6（tranfer时，lc、hc=0两个计数器分别++记录原bin、新binTreeNode数量，<=UNTREEIFY_THRESHOLD 则untreeify(lo)）
static final int UNTREEIFY_THRESHOLD = 6;

//
static final int MIN_TREEIFY_CAPACITY = 64;

//
private static final int MIN_TRANSFER_STRIDE = 16;

//
private static int RESIZE_STAMP_BITS = 16;

// 2^15-1，help resize的最大线程数
private static final int MAX_RESIZERS = (1 << (32 - RESIZE_STAMP_BITS)) - 1;

// 32-16=16，sizeCtl中记录size大小的偏移量
private static final int RESIZE_STAMP_SHIFT = 32 - RESIZE_STAMP_BITS;
```

```
// forwarding nodes的hash值
static final int MOVED     = -1;

// 树根节点的hash值
static final int TREEBIN   = -2;

// ReservationNode的hash值
static final int RESERVED  = -3;

// 可用处理器数量
static final int NCPU = Runtime.getRuntime().availableProcessors();
```

上面是ConcurrentHashMap定义的常量，简单易懂，就不多阐述了。下面介绍ConcurrentHashMap几个很重要的概念。

1. **table**：用来存放Node节点数据的，默认为null，默认大小为16的数组，每次扩容时大小总是2的幂次方；
2. **nextTable**：扩容时新生成的数据，数组为table的两倍；
3. **Node**：节点，保存key-value的数据结构；
4. **ForwardingNode**：一个特殊的Node节点，hash值为-1，其中存储nextTable的引用。只有table发生扩容的时候，ForwardingNode才会发挥作用，作为一个占位符放在table中表示当前节点为null或则已经被移动
5. sizeCtl

   ：控制标识符，用来控制table初始化和扩容操作的，在不同的地方有不同的用途，其值也不同，所代表的含义也不同

   - 负数代表正在进行初始化或扩容操作
   - -1代表正在初始化
   - -N 表示有N-1个线程正在进行扩容操作
   - 正数或0代表hash表还没有被初始化，这个数值表示初始化或下一次进行扩容的大小

重要内部类

为了实现ConcurrentHashMap，Doug Lea提供了许多内部类来进行辅助实现，如Node，TreeNode,TreeBin等等。下面我们就一起来看看ConcurrentHashMap几个重要的内部类。

Node

作为ConcurrentHashMap中最核心、最重要的内部类，Node担负着重要角色：key-value键值对。所有插入ConCurrentHashMap的中数据都将会包装在Node中。定义如下：

```
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        volatile V val;              //带有volatile，保证可见性
        volatile Node<K,V> next;     //下一个节点的指针

        Node(int hash, K key, V val, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.val = val;
            this.next = next;
        }
```

```
        public final K getKey()       { return key; }
        public final V getValue()     { return val; }
        public final int hashCode()   { return key.hashCode() ^ val.hashCode();
}
        public final String toString(){ return key + "=" + val; }
        /** 不允许修改value的值 */
        public final V setValue(V value) {
            throw new UnsupportedOperationException();
        }

        public final boolean equals(Object o) {
            Object k, v, u; Map.Entry<?,?> e;
            return ((o instanceof Map.Entry) &&
                    (k = (e = (Map.Entry<?,?>)o).getKey()) != null &&
                    (v = e.getValue()) != null &&
                    (k == key || k.equals(key)) &&
                    (v == (u = val) || v.equals(u)));
        }

        /**  赋值get()方法 */
        Node<K,V> find(int h, Object k) {
            Node<K,V> e = this;
            if (k != null) {
                do {
                    K ek;
                    if (e.hash == h &&
                            ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                } while ((e = e.next) != null);
            }
            return null;
        }
    }
```

在Node内部类中，其属性value、next都是带有volatile的。同时其对value的setter方法进行了特殊处理，不允许直接调用其setter方法来修改value的值。最后Node还提供了find方法来赋值map.get()。

TreeNode

我们在学习HashMap的时候就知道，HashMap的核心数据结构就是链表。在ConcurrentHashMap中就不一样了，如果链表的数据过长是会转换为红黑树来处理。当它并不是直接转换，而是将这些链表的节点包装成TreeNode放在TreeBin对象中，然后由TreeBin完成红黑树的转换。所以TreeNode也必须是ConcurrentHashMap的一个核心类，其为树节点类，定义如下：

```
    static final class TreeNode<K,V> extends Node<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;

        TreeNode(int hash, K key, V val, Node<K,V> next,
                 TreeNode<K,V> parent) {
            super(hash, key, val, next);
            this.parent = parent;
        }
```

```
        Node<K,V> find(int h, Object k) {
            return findTreeNode(h, k, null);
        }

        //查找hash为h，key为k的节点
        final TreeNode<K,V> findTreeNode(int h, Object k, Class<?> kc) {
            if (k != null) {
                TreeNode<K,V> p = this;
                do  {
                    int ph, dir; K pk; TreeNode<K,V> q;
                    TreeNode<K,V> pl = p.left, pr = p.right;
                    if ((ph = p.hash) > h)
                        p = pl;
                    else if (ph < h)
                        p = pr;
                    else if ((pk = p.key) == k || (pk != null && k.equals(pk)))
                        return p;
                    else if (pl == null)
                        p = pr;
                    else if (pr == null)
                        p = pl;
                    else if ((kc != null ||
                            (kc = comparableClassFor(k)) != null) &&
                            (dir = compareComparables(kc, k, pk)) != 0)
                        p = (dir < 0) ? pl : pr;
                    else if ((q = pr.findTreeNode(h, k, kc)) != null)
                        return q;
                    else
                        p = pl;
                } while (p != null);
            }
            return null;
        }
    }
```

源码展示TreeNode继承Node，且提供了findTreeNode用来查找查找hash为h，key为k的节点。

TreeBin

该类并不负责key-value的键值对包装，它用于在链表转换为红黑树时包装TreeNode节点，也就是说ConcurrentHashMap红黑树存放是TreeBin，不是TreeNode。该类封装了一系列的方法，包括putTreeVal、lookRoot、UNlookRoot、remove、balanceInsetion、balanceDeletion。由于TreeBin的代码太长我们这里只展示构造方法（构造方法就是构造红黑树的过程）：

```
    static final class TreeBin<K,V> extends Node<K,V> {
        TreeNode<K, V> root;
        volatile TreeNode<K, V> first;
        volatile Thread waiter;
        volatile int lockState;
        static final int WRITER = 1; // set while holding write lock
        static final int WAITER = 2; // set when waiting for write lock
        static final int READER = 4; // increment value for setting read lock

        TreeBin(TreeNode<K, V> b) {
            super(TREEBIN, null, null, null);
```

```
            this.first = b;
            TreeNode<K, V> r = null;
            for (TreeNode<K, V> x = b, next; x != null; x = next) {
                next = (TreeNode<K, V>) x.next;
                x.left = x.right = null;
                if (r == null) {
                    x.parent = null;
                    x.red = false;
                    r = x;
                } else {
                    K k = x.key;
                    int h = x.hash;
                    Class<?> kc = null;
                    for (TreeNode<K, V> p = r; ; ) {
                        int dir, ph;
                        K pk = p.key;
                        if ((ph = p.hash) > h)
                            dir = -1;
                        else if (ph < h)
                            dir = 1;
                        else if ((kc == null &&
                                (kc = comparableClassFor(k)) == null) ||
                                (dir = compareComparables(kc, k, pk)) == 0)
                            dir = tieBreakOrder(k, pk);
                        TreeNode<K, V> xp = p;
                        if ((p = (dir <= 0) ? p.left : p.right) == null) {
                            x.parent = xp;
                            if (dir <= 0)
                                xp.left = x;
                            else
                                xp.right = x;
                            r = balanceInsertion(r, x);
                            break;
                        }
                    }
                }
            }
            this.root = r;
            assert checkInvariants(root);
        }

        /** 省略很多代码 */
    }
```

通过构造方法是不是发现了部分端倪，构造方法就是在构造一个红黑树的过程。

ForwardingNode

这是一个真正的辅助类，该类仅仅只存活在ConcurrentHashMap扩容操作时。只是一个标志节点，并且指向nextTable，它提供find方法而已。该类也是集成Node节点，其hash为-1，key、value、next均为null。如下：

```
    static final class ForwardingNode<K,V> extends Node<K,V> {
        final Node<K,V>[] nextTable;
        ForwardingNode(Node<K,V>[] tab) {
            super(MOVED, null, null, null);
            this.nextTable = tab;
```

```
        }

        Node<K,V> find(int h, Object k) {
            // loop to avoid arbitrarily deep recursion on forwarding nodes
            outer: for (Node<K,V>[] tab = nextTable;;) {
                Node<K,V> e; int n;
                if (k == null || tab == null || (n = tab.length) == 0 ||
                        (e = tabAt(tab, (n - 1) & h)) == null)
                    return null;
                for (;;) {
                    int eh; K ek;
                    if ((eh = e.hash) == h &&
                            ((ek = e.key) == k || (ek != null && k.equals(ek))))
                        return e;
                    if (eh < 0) {
                        if (e instanceof ForwardingNode) {
                            tab = ((ForwardingNode<K,V>)e).nextTable;
                            continue outer;
                        }
                        else
                            return e.find(h, k);
                    }
                    if ((e = e.next) == null)
                        return null;
                }
            }
        }
    }
```

构造函数

ConcurrentHashMap提供了一系列的构造函数用于创建ConcurrentHashMap对象：

```
    public ConcurrentHashMap() {
    }

    public ConcurrentHashMap(int initialCapacity) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException();
        int cap = ((initialCapacity >= (MAXIMUM_CAPACITY >>> 1)) ?
                   MAXIMUM_CAPACITY :
                   tableSizeFor(initialCapacity + (initialCapacity >>> 1) + 1));
        this.sizeCtl = cap;
    }

    public ConcurrentHashMap(Map<? extends K, ? extends V> m) {
        this.sizeCtl = DEFAULT_CAPACITY;
        putAll(m);
    }

    public ConcurrentHashMap(int initialCapacity, float loadFactor) {
        this(initialCapacity, loadFactor, 1);
    }

    public ConcurrentHashMap(int initialCapacity,
                             float loadFactor, int concurrencyLevel) {
```

```
        if (!(loadFactor > 0.0f) || initialCapacity < 0 || concurrencyLevel <=
0)
            throw new IllegalArgumentException();
        if (initialCapacity < concurrencyLevel)   // Use at least as many bins
            initialCapacity = concurrencyLevel;   // as estimated threads
        long size = (long)(1.0 + (long)initialCapacity / loadFactor);
        int cap = (size >= (long)MAXIMUM_CAPACITY) ?
            MAXIMUM_CAPACITY : tableSizeFor((int)size);
        this.sizeCtl = cap;
    }
```

初始化： initTable()

ConcurrentHashMap的初始化主要由initTable()方法实现，在上面的构造函数中我们可以看到，其实ConcurrentHashMap在构造函数中并没有做什么事，仅仅只是设置了一些参数而已。其真正的初始化是发生在插入的时候，例如put、merge、compute、computeIfAbsent、computeIfPresent操作时。其方法定义如下：

```
    private final Node<K,V>[] initTable() {
        Node<K,V>[] tab; int sc;
        while ((tab = table) == null || tab.length == 0) {
            //sizeCtl < 0 表示有其他线程在初始化，该线程必须挂起
            if ((sc = sizeCtl) < 0)
                Thread.yield();
            // 如果该线程获取了初始化的权利，则用CAS将sizeCtl设置为-1，表示本线程正在初始
化
            else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {
                    // 进行初始化
                try {
                    if ((tab = table) == null || tab.length == 0) {
                        int n = (sc > 0) ? sc : DEFAULT_CAPACITY;
                        @SuppressWarnings("unchecked")
                        Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];
                        table = tab = nt;
                        // 下次扩容的大小
                        sc = n - (n >>> 2); ///相当于0.75*n 设置一个扩容的阈值
                    }
                } finally {
                    sizeCtl = sc;
                }
                break;
            }
        }
        return tab;
    }
```

初始化方法initTable()的关键就在于sizeCtl，该值默认为0，如果在构造函数时有参数传入该值则为2的幂次方。该值如果 < 0，表示有其他线程正在初始化，则必须暂停该线程。如果线程获得了初始化的权限则先将sizeCtl设置为-1，防止有其他线程进入，最后将sizeCtl设置0.75 * n，表示扩容的阈值。

put操作

ConcurrentHashMap最常用的put、get操作，ConcurrentHashMap的put操作与HashMap并没有多大区别，其核心思想依然是根据hash值计算节点插入在table的位置，如果该位置为空，则直接插入，否则插入到链表或者树中。但是ConcurrentHashMap会涉及到多线程情况就会复杂很多。我们先看源代码，然后根据源代码一步一步分析：

```
    public V put(K key, V value) {
        return putVal(key, value, false);
    }

    final V putVal(K key, V value, boolean onlyIfAbsent) {
        //key、value均不能为null
        if (key == null || value == null) throw new NullPointerException();
        //计算hash值
        int hash = spread(key.hashCode());
        int binCount = 0;
        for (Node<K,V>[] tab = table;;) {
            Node<K,V> f; int n, i, fh;
            // table为null，进行初始化工作
            if (tab == null || (n = tab.length) == 0)
                tab = initTable();
            //如果i位置没有节点，则直接插入，不需要加锁
            else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
                if (casTabAt(tab, i, null,
                        new Node<K,V>(hash, key, value, null)))
                    break;                   // no lock when adding to empty bin
            }
            // 有线程正在进行扩容操作，则先帮助扩容
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
            else {
                V oldVal = null;
                //对该节点进行加锁处理（hash值相同的链表的头节点），对性能有点儿影响
                synchronized (f) {
                    if (tabAt(tab, i) == f) {
                        //fh > 0 表示为链表，将该节点插入到链表尾部
                        if (fh >= 0) {
                            binCount = 1;
                            for (Node<K,V> e = f;; ++binCount) {
                                K ek;
                                //hash 和 key 都一样，替换value
                                if (e.hash == hash &&
                                        ((ek = e.key) == key ||
                                                (ek != null && key.equals(ek))))
{
                                    oldVal = e.val;
                                    //putIfAbsent()
                                    if (!onlyIfAbsent)
                                        e.val = value;
                                    break;
                                }
                                Node<K,V> pred = e;
                                //链表尾部   直接插入
                                if ((e = e.next) == null) {
                                    pred.next = new Node<K,V>(hash, key,
                                            value, null);
                                    break;
                                }
```

```
                            }
                        }
                        //树节点，按照树的插入操作进行插入
                        else if (f instanceof TreeBin) {
                            Node<K,V> p;
                            binCount = 2;
                            if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
                                    value)) != null) {
                                oldVal = p.val;
                                if (!onlyIfAbsent)
                                    p.val = value;
                            }
                        }
                    }
                }
                if (binCount != 0) {
                    // 如果链表长度已经达到临界值8 就需要把链表转换为树结构
                    if (binCount >= TREEIFY_THRESHOLD)
                        treeifyBin(tab, i);
                    if (oldVal != null)
                        return oldVal;
                    break;
                }
            }
        }

        //size + 1
        addCount(1L, binCount);
        return null;
    }
```

按照上面的源码，我们可以确定put整个流程如下：

- 判空；ConcurrentHashMap的key、value都不允许为null
- 计算hash。利用方法计算hash值。

```
    static final int spread(int h) {
        return (h ^ (h >>> 16)) & HASH_BITS;
    }
```

- 遍历table，进行节点插入操作，过程如下：
    - 如果table为空，则表示ConcurrentHashMap还没有初始化，则进行初始化操作：initTable()
    - 根据hash值获取节点的位置i，若该位置为空，则直接插入，这个过程是不需要加锁的。计算f位置：i=(n - 1) & hash
    - 如果检测到fh = f.hash == -1，则f是ForwardingNode节点，表示有其他线程正在进行扩容操作，则帮助线程一起进行扩容操作
    - 如果f.hash >= 0 表示是链表结构，则遍历链表，如果存在当前key节点则替换value，否则插入到链表尾部。如果f是TreeBin类型节点，则按照红黑树的方法更新或者增加节点
    - 若链表长度 > TREEIFY_THRESHOLD(默认是8)，则将链表转换为红黑树结构
- 调用addCount方法，ConcurrentHashMap的size + 1

这里整个put操作已经完成。

get操作

ConcurrentHashMap的get操作还是挺简单的，无非就是通过hash来找key相同的节点而已，当然需要区分链表和树形两种情况。

```
    public V get(Object key) {
        Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;
        // 计算hash
        int h = spread(key.hashCode());
        if ((tab = table) != null && (n = tab.length) > 0 &&
                (e = tabAt(tab, (n - 1) & h)) != null) {
            // 搜索到的节点key与传入的key相同且不为null,直接返回这个节点
            if ((eh = e.hash) == h) {
                if ((ek = e.key) == key || (ek != null && key.equals(ek)))
                    return e.val;
            }
            // 树
            else if (eh < 0)
                return (p = e.find(h, key)) != null ? p.val : null;
            // 链表，遍历
            while ((e = e.next) != null) {
                if (e.hash == h &&
                        ((ek = e.key) == key || (ek != null && key.equals(ek))))
                    return e.val;
            }
        }
        return null;
    }
```

get操作的整个逻辑非常清楚：

- 计算hash值
- 判断table是否为空，如果为空，直接返回null
- 根据hash值获取table中的Node节点（tabAt(tab, (n - 1) & h)），然后根据链表或者树形方式找到相对应的节点，返回其value值。

size 操作

ConcurrentHashMap的size()方法我们虽然用得不是很多，但是我们还是很有必要去了解的。ConcurrentHashMap的size()方法返回的是一个不精确的值，因为在进行统计的时候有其他线程正在进行插入和删除操作。当然为了这个不精确的值，ConcurrentHashMap也是操碎了心。 为了更好地统计size，ConcurrentHashMap提供了baseCount、counterCells两个辅助变量和一个CounterCell辅助内部类。

```
    @sun.misc.Contended static final class CounterCell {
        volatile long value;
        CounterCell(long x) { value = x; }
    }

    //ConcurrentHashMap中元素个数,但返回的不一定是当前Map的真实元素个数。基于CAS无锁更新
    private transient volatile long baseCount;

    private transient volatile CounterCell[] counterCells;
```

这里我们需要清楚CounterCell 的定义 size()方法定义如下：

```
    public int size() {
        long n = sumCount();
        return ((n < 0L) ? 0 :
                (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
                (int)n);
    }
```

内部调用sunmCount()：

```
    final long sumCount() {
        CounterCell[] as = counterCells; CounterCell a;
        long sum = baseCount;
        if (as != null) {
            for (int i = 0; i < as.length; ++i) {
                //遍历，所有counter求和
                if ((a = as[i]) != null)
                    sum += a.value;
            }
        }
        return sum;
    }
```

sumCount()就是迭代counterCells来统计sum的过程。我们知道put操作时，肯定会影响size()，我们就来看看CouncurrentHashMap是如何为了这个不和谐的size()操碎了心。 在put()方法最后会调用addCount()方法，该方法主要做两件事，一件更新baseCount的值，第二件检测是否进行扩容，我们只看更新baseCount部分：

```
    private final void addCount(long x, int check) {
        CounterCell[] as; long b, s;
        // s = b + x，完成baseCount++操作；
        if ((as = counterCells) != null ||
            !U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)) {
            CounterCell a; long v; int m;
            boolean uncontended = true;
            if (as == null || (m = as.length - 1) < 0 ||
                (a = as[ThreadLocalRandom.getProbe() & m]) == null ||
                !(uncontended =
                  U.compareAndSwapLong(a, CELLVALUE, v = a.value, v + x))) {
                //  多线程CAS发生失败时执行
                fullAddCount(x, uncontended);
                return;
            }
            if (check <= 1)
                return;
            s = sumCount();
        }

        // 检查是否进行扩容
    }
```

x == 1，如果counterCells == null，则U.compareAndSwapLong(this, BASECOUNT, b = baseCount, s = b + x)，如果并发竞争比较大可能会导致改过程失败，如果失败则最终会调用fullAddCount()方法。其实为了提高高并发的时候baseCount可见性的失败问题，又避免一直重试，JDK 8 引入了类Striped64，其中LongAdder和DoubleAdder都是基于该类实现的，而CounterCell也是基于Striped64实现的。如果counterCells ！= null，且uncontended = U.compareAndSwapLong(a, CELLVALUE, v = a.value, v +

x)也失败了，同样会调用fullAddCount()方法，最后调用sumCount()计算s。 其实在1.8中，它不推荐size()方法，而是推崇mappingCount()方法，该方法的定义和size()方法基本一致：

```
    public long mappingCount() {
        long n = sumCount();
        return (n < 0L) ? 0L : n; // ignore transient negative values
    }
```

扩容操作

当ConcurrentHashMap中table元素个数达到了容量阈值（sizeCtl）时，则需要进行扩容操作。在put操作时最后一个会调用addCount(long x, int check)，该方法主要做两个工作：1.更新baseCount；2.检测是否需要扩容操作。如下：

```
    private final void addCount(long x, int check) {
        CounterCell[] as; long b, s;
        // 更新baseCount

        //check >= 0 :则需要进行扩容操作
        if (check >= 0) {
            Node<K,V>[] tab, nt; int n, sc;
            while (s >= (long)(sc = sizeCtl) && (tab = table) != null &&
                    (n = tab.length) < MAXIMUM_CAPACITY) {
                int rs = resizeStamp(n);
                if (sc < 0) {
                    if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                            sc == rs + MAX_RESIZERS || (nt = nextTable) == null
||
                            transferIndex <= 0)
                        break;
                    if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1))
                        transfer(tab, nt);
                }

                //当前线程是唯一的或是第一个发起扩容的线程  此时nextTable=null
                else if (U.compareAndSwapInt(this, SIZECTL, sc,
                        (rs << RESIZE_STAMP_SHIFT) + 2))
                    transfer(tab, null);
                s = sumCount();
            }
        }
    }
```

transfer()方法为ConcurrentHashMap扩容操作的核心方法。由于ConcurrentHashMap支持多线程扩容，而且也没有进行加锁，所以实现会变得有点儿复杂。整个扩容操作分为两步：

1. 构建一个nextTable，其大小为原来大小的两倍，这个步骤是在单线程环境下完成的
2. 将原来table里面的内容复制到nextTable中，这个步骤是允许多线程操作的，所以性能得到提升，减少了扩容的时间消耗

我们先来看看源代码，然后再一步一步分析：

```
    private final void transfer(Node<K,V>[] tab, Node<K,V>[] nextTab) {
        int n = tab.length, stride;
        // 每核处理的量小于16，则强制赋值16
        if ((stride = (NCPU > 1) ? (n >>> 3) / NCPU : n) < MIN_TRANSFER_STRIDE)
```

```
            stride = MIN_TRANSFER_STRIDE; // subdivide range
        if (nextTab == null) {            // initiating
            try {
                @SuppressWarnings("unchecked")
                Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n << 1];        //构
建一个nextTable对象，其容量为原来容量的两倍
                nextTab = nt;
            } catch (Throwable ex) {      // try to cope with OOME
                sizeCtl = Integer.MAX_VALUE;
                return;
            }
            nextTable = nextTab;
            transferIndex = n;
        }
        int nextn = nextTab.length;
        // 连接点指针，用于标志位（fwd的hash值为-1，fwd.nextTable=nextTab）
        ForwardingNode<K,V> fwd = new ForwardingNode<K,V>(nextTab);
        // 当advance == true时，表明该节点已经处理过了
        boolean advance = true;
        boolean finishing = false; // to ensure sweep before committing nextTab
        for (int i = 0, bound = 0;;) {
            Node<K,V> f; int fh;
            // 控制 --i ,遍历原hash表中的节点
            while (advance) {
                int nextIndex, nextBound;
                if (--i >= bound || finishing)
                    advance = false;
                else if ((nextIndex = transferIndex) <= 0) {
                    i = -1;
                    advance = false;
                }
                // 用CAS计算得到的transferIndex
                else if (U.compareAndSwapInt
                        (this, TRANSFERINDEX, nextIndex,
                                nextBound = (nextIndex > stride ?
                                        nextIndex - stride : 0))) {
                    bound = nextBound;
                    i = nextIndex - 1;
                    advance = false;
                }
            }
            if (i < 0 || i >= n || i + n >= nextn) {
                int sc;
                // 已经完成所有节点复制了
                if (finishing) {
                    nextTable = null;
                    table = nextTab;          // table 指向nextTable
                    sizeCtl = (n << 1) - (n >>> 1);     // sizeCtl阈值为原来的1.5
倍
                    return;      // 跳出死循环，
                }
                // CAS 更扩容阈值，在这里面sizectl值减一，说明新加入一个线程参与到扩容操
作
                if (U.compareAndSwapInt(this, SIZECTL, sc = sizeCtl, sc - 1)) {
                    if ((sc - 2) != resizeStamp(n) << RESIZE_STAMP_SHIFT)
                        return;
                    finishing = advance = true;
                    i = n; // recheck before commit
```

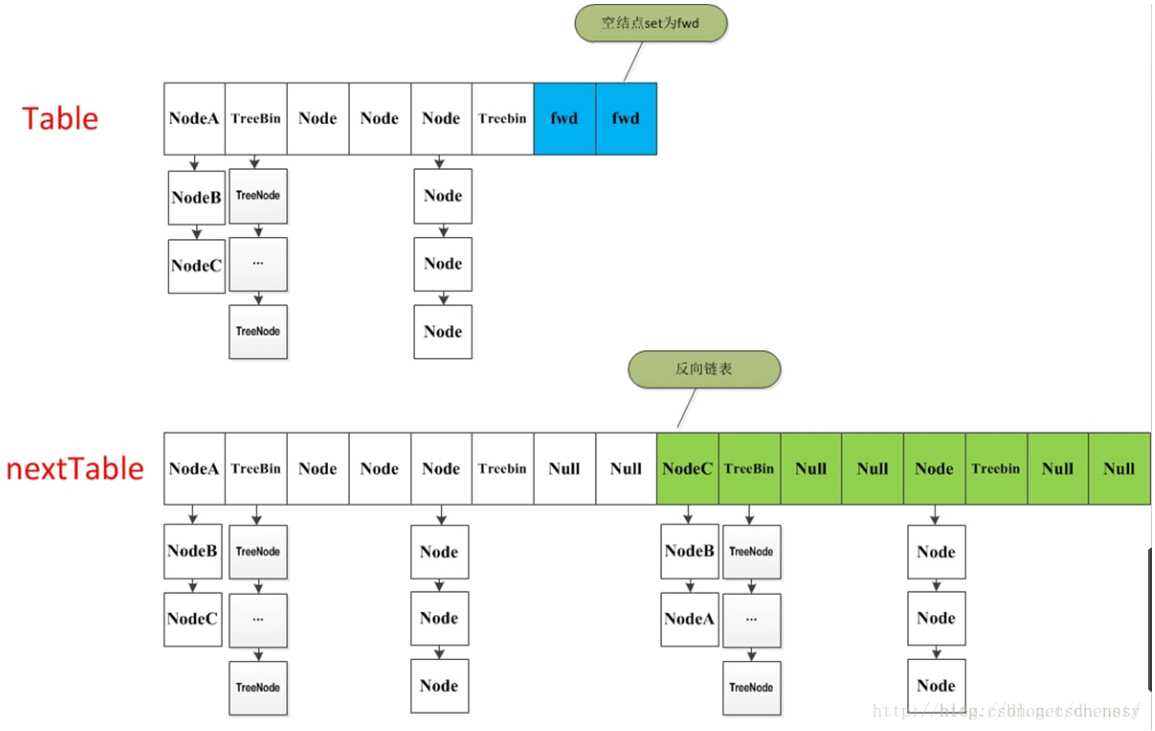
```
                }
            }
            // 遍历的节点为null，则放入到ForwardingNode 指针节点
            else if ((f = tabAt(tab, i)) == null)
                advance = casTabAt(tab, i, null, fwd);
            // f.hash == -1 表示遍历到了ForwardingNode节点，意味着该节点已经处理过了
            // 这里是控制并发扩容的核心
            else if ((fh = f.hash) == MOVED)
                advance = true; // already processed
            else {
                // 节点加锁
                synchronized (f) {
                    // 节点复制工作
                    if (tabAt(tab, i) == f) {
                        Node<K,V> ln, hn;
                        // fh >= 0 ,表示为链表节点
                        if (fh >= 0) {
                            // 构造两个链表  一个是原链表  另一个是原链表的反序排列
                            int runBit = fh & n;
                            Node<K,V> lastRun = f;
                            for (Node<K,V> p = f.next; p != null; p = p.next) {
                                int b = p.hash & n;
                                if (b != runBit) {
                                    runBit = b;
                                    lastRun = p;
                                }
                            }
                            if (runBit == 0) {
                                ln = lastRun;
                                hn = null;
                            }
                            else {
                                hn = lastRun;
                                ln = null;
                            }
                            for (Node<K,V> p = f; p != lastRun; p = p.next) {
                                int ph = p.hash; K pk = p.key; V pv = p.val;
                                if ((ph & n) == 0)
                                    ln = new Node<K,V>(ph, pk, pv, ln);
                                else
                                    hn = new Node<K,V>(ph, pk, pv, hn);
                            }
                            // 在nextTable i 位置处插上链表
                            setTabAt(nextTab, i, ln);
                            // 在nextTable i + n 位置处插上链表
                            setTabAt(nextTab, i + n, hn);
                            // 在table i 位置处插上ForwardingNode 表示该节点已经处理
过了
                            setTabAt(tab, i, fwd);
                            // advance = true 可以执行--i动作，遍历节点
                            advance = true;
                        }
                        // 如果是TreeBin，则按照红黑树进行处理，处理逻辑与上面一致
                        else if (f instanceof TreeBin) {
                            TreeBin<K,V> t = (TreeBin<K,V>)f;
                            TreeNode<K,V> lo = null, loTail = null;
                            TreeNode<K,V> hi = null, hiTail = null;
                            int lc = 0, hc = 0;
```

```
                            for (Node<K,V> e = t.first; e != null; e = e.next) {
                                int h = e.hash;
                                TreeNode<K,V> p = new TreeNode<K,V>
                                        (h, e.key, e.val, null, null);
                                if ((h & n) == 0) {
                                    if ((p.prev = loTail) == null)
                                        lo = p;
                                    else
                                        loTail.next = p;
                                    loTail = p;
                                    ++lc;
                                }
                                else {
                                    if ((p.prev = hiTail) == null)
                                        hi = p;
                                    else
                                        hiTail.next = p;
                                    hiTail = p;
                                    ++hc;
                                }
                            }

                            // 扩容后树节点个数若<=6，将树转链表
                            ln = (lc <= UNTREEIFY_THRESHOLD) ? untreeify(lo) :
                                    (hc != 0) ? new TreeBin<K,V>(lo) : t;
                            hn = (hc <= UNTREEIFY_THRESHOLD) ? untreeify(hi) :
                                    (lc != 0) ? new TreeBin<K,V>(hi) : t;
                            setTabAt(nextTab, i, ln);
                            setTabAt(nextTab, i + n, hn);
                            setTabAt(tab, i, fwd);
                            advance = true;
                        }
                    }
                }
            }
        }
    }
```

上面的源码有点儿长，稍微复杂了一些，在这里我们抛弃它多线程环境，我们从单线程角度来看：

1. 为每个内核分任务，并保证其不小于16
2. 检查nextTable是否为null，如果是，则初始化nextTable，使其容量为table的两倍
3. 死循环遍历节点，知道finished：节点从table复制到nextTable中，支持并发，请思路如下：
    - 如果节点 f 为null，则插入ForwardingNode（采用Unsafe.compareAndSwapObjectf方法实现），这个是触发并发扩容的关键
    - 如果f为链表的头节点（fh >= 0）,则先构造一个反序链表，然后把他们分别放在nextTable的i和i + n位置，并将ForwardingNode 插入原节点位置，代表已经处理过了
    - 如果f为TreeBin节点，同样也是构造一个反序 ，同时需要判断是否需要进行unTreeify()操作，并把处理的结果分别插入到nextTable的i 和i+nw位置，并插入ForwardingNode 节点
4. 所有节点复制完成后，则将table指向nextTable，同时更新sizeCtl = nextTable的0.75倍，完成扩容过程

在多线程环境下，ConcurrentHashMap用两点来保证正确性：ForwardingNode和synchronized。当一个线程遍历到的节点如果是ForwardingNode，则继续往后遍历，如果不是，则将该节点加锁，防止其他线程进入，完成后设置ForwardingNode节点，以便要其他线程可以看到该节点已经处理过了，如此交叉进行，高效而又安全。 下图是扩容的过程（来自：http://blog.csdn.net/u010723709/article/details/48007881 ）：



在put操作时如果发现fh.hash = -1，则表示正在进行扩容操作，则当前线程会协助进行扩容操作。

```
            else if ((fh = f.hash) == MOVED)
                tab = helpTransfer(tab, f);
```

helpTransfer()方法为协助扩容方法，当调用该方法的时候，nextTable一定已经创建了，所以该方法主要则是进行复制工作。如下：

```
    final Node<K,V>[] helpTransfer(Node<K,V>[] tab, Node<K,V> f) {
        Node<K,V>[] nextTab; int sc;
        if (tab != null && (f instanceof ForwardingNode) &&
                (nextTab = ((ForwardingNode<K,V>)f).nextTable) != null) {
            int rs = resizeStamp(tab.length);
            while (nextTab == nextTable && table == tab &&
                    (sc = sizeCtl) < 0) {
                if ((sc >>> RESIZE_STAMP_SHIFT) != rs || sc == rs + 1 ||
                        sc == rs + MAX_RESIZERS || transferIndex <= 0)
                    break;
                if (U.compareAndSwapInt(this, SIZECTL, sc, sc + 1)) {
                    transfer(tab, nextTab);
                    break;
                }
            }
            return nextTab;
        }
        return table;
    }
```

转换红黑树

在put操作是，如果发现链表结构中的元素超过了TREEIFY_THRESHOLD（默认为8），则会把链表转换为红黑树，已便于提高查询效率。如下：

```
if (binCount >= TREEIFY_THRESHOLD)
    treeifyBin(tab, i);
```

调用treeifyBin方法用与将链表转换为红黑树。

```
private final void treeifyBin(Node<K,V>[] tab, int index) {
        Node<K,V> b; int n, sc;
        if (tab != null) {
            if ((n = tab.length) < MIN_TREEIFY_CAPACITY)//如果table.length<64 就扩大一倍 返回
                tryPresize(n << 1);
            else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
                synchronized (b) {
                    if (tabAt(tab, index) == b) {
                        TreeNode<K,V> hd = null, tl = null;
                        //构造了一个TreeBin对象 把所有Node节点包装成TreeNode放进去
                        for (Node<K,V> e = b; e != null; e = e.next) {
                            TreeNode<K,V> p =
                                new TreeNode<K,V>(e.hash, e.key, e.val,
                                                  null, null);//这里只是利用了TreeNode封装 而没有利用TreeNode的next域和parent域
                            if ((p.prev = tl) == null)
                                hd = p;
                            else
                                tl.next = p;
                            tl = p;
                        }
                        //在原来index的位置 用TreeBin替换掉原来的Node对象
                        setTabAt(tab, index, new TreeBin<K,V>(hd));
                    }
                }
            }
        }
    }
```

从上面源码可以看出，构建红黑树的过程是同步的，进入同步后过程如下：

1. 根据table中index位置Node链表，重新生成一个hd为头结点的TreeNode
2. 根据hd头结点，生成TreeBin树结构，并用TreeBin替换掉原来的Node对象。

ConcurrentHashMap 红黑树的构建过程

在put操作时，如果发现链表结构中的元素超过了TREEIFY_THRESHOLD（默认为8），则会把链表转换为红黑树，已便于提高查询效率。代码如下：

```
if (binCount >= TREEIFY_THRESHOLD)
    treeifyBin(tab, i);
```

下面博主将详细分析整个过程，并用一个链表转换为红黑树的过程为案例来分析。博文从如下几个方法进行分析阐述：

1. 红黑树
2. ConcurrentHashMap链表转红黑树源码分析

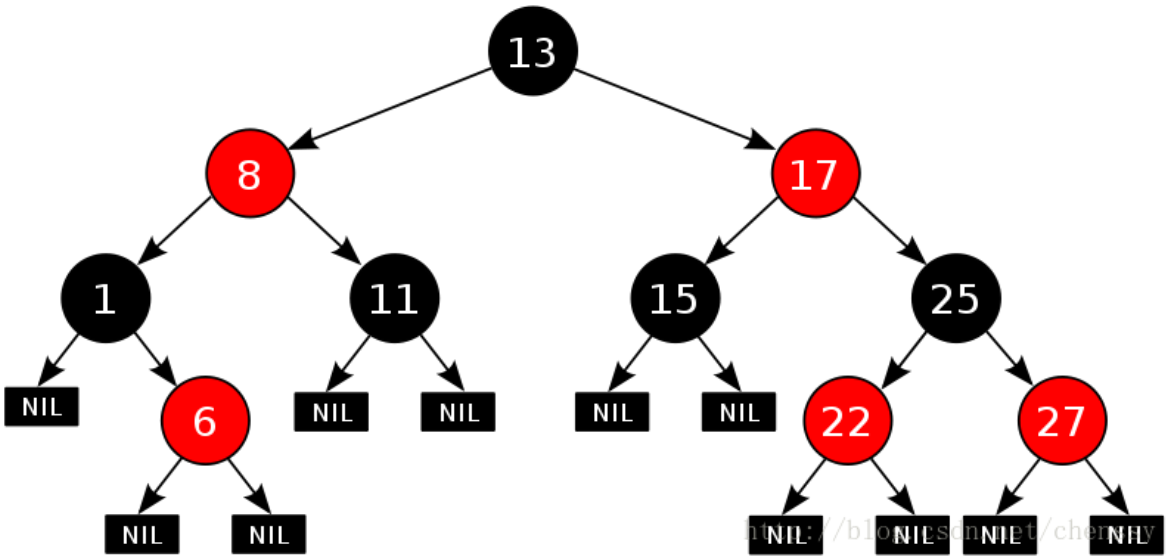
3. 链表转红黑树案例

红黑树

先看红黑树的基本概念：红黑树是一课特殊的平衡二叉树，主要用它存储有序的数据，提供高效的数据检索，时间复杂度为O(lgn)。红黑树每个节点都有一个标识位表示颜色，红色或黑色，具备五种特性：

1. 每个节点非红即黑
2. 根节点为黑色
3. 每个叶子节点为黑色。叶子节点为NIL节点，即空节点
4. 如果一个节点为红色，那么它的子节点一定是黑色
5. 从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点

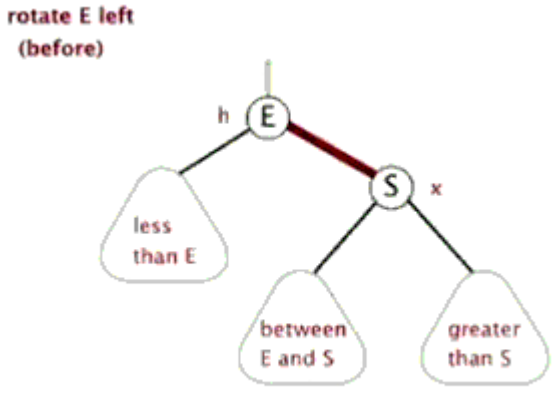
请牢记这五个特性，它在维护红黑树时选的格外重要 红黑树结构图如下：



对于红黑树而言，它主要包括三个步骤：左旋、右旋、着色。所有不符合上面五个特性的“红黑树”都可以通过这三个步骤调整为正规的红黑树。

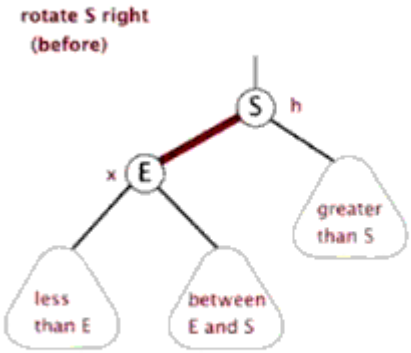
旋转

当对红黑树进行插入和删除操作时可能会破坏红黑树的特性。为了继续保持红黑树的性质，则需要通过对红黑树进行旋转和重新着色处理，其中旋转包括左旋、右旋。 **左旋** 左旋示意图如下：



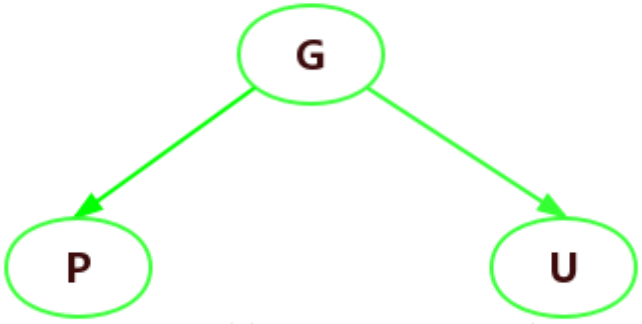
左旋处理过程比较简单，将E的右孩子S调整

为E的父节点、S节点的左孩子作为调整后E节点的右孩子。 **右旋**

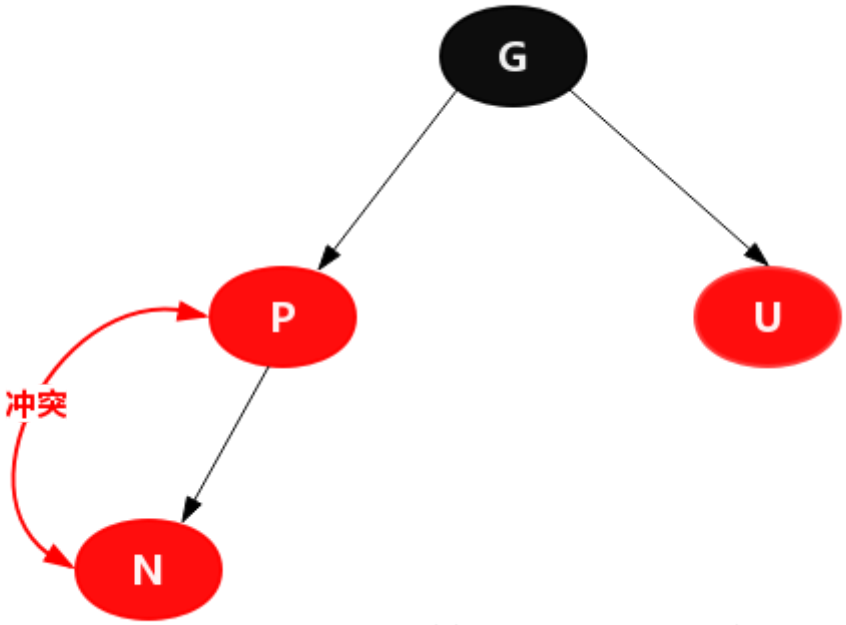


红黑树插入节点

由于链表转换为红黑树只有添加操作，加上篇幅有限所以这里就只介绍红黑树的插入操作，关于红黑树的详细情况，烦请各位Google。 在分析过程中，我们已下面一颗简单的树为案例，根节点G、有两个子节点P、U，我们新增的节点为N

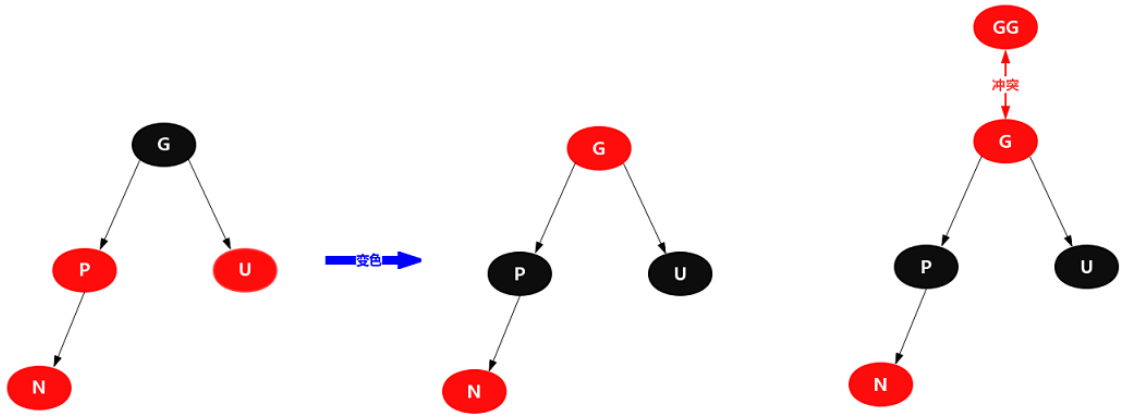


红黑树默认插入的节点为红色，因为如果为黑色，则一定会破坏红黑树的规则5（从一个节点到该节点的子孙节点的所有路径包含相同个数的黑色节点）。尽管默认的节点为红色，插入之后也会导致红黑树失衡。红黑树插入操作导致其失衡的主要原因在于插入的当前节点与其父节点的颜色冲突导致（红红，违背规则4：如果一个节点为红色，那么它的子节点一定是黑色）。



要解决这

类冲突就靠上面三个操作：左旋、右旋、重新着色。由于是红红冲突，那么其祖父节点一定存在且为黑色，但是叔父节点U颜色不确定，根据叔父节点的颜色则可以做相应的调整。 **1 叔父U节点是红色** 如果叔父节点为红色，那么处理过程则变得比较简单了：更换G与P、U节点的颜色，下图（一）。

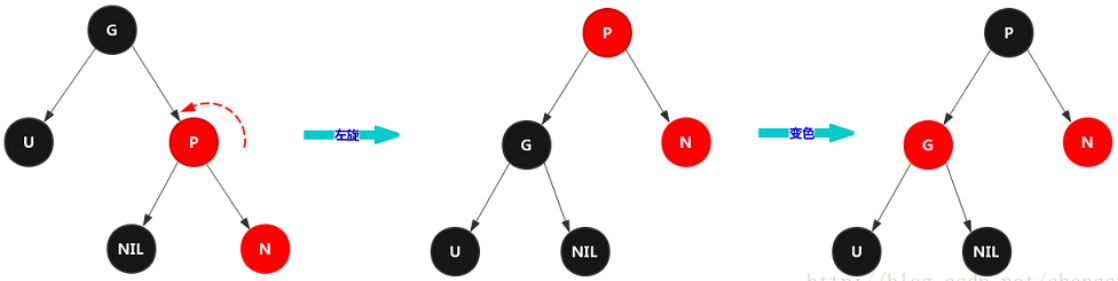


图一　　图二

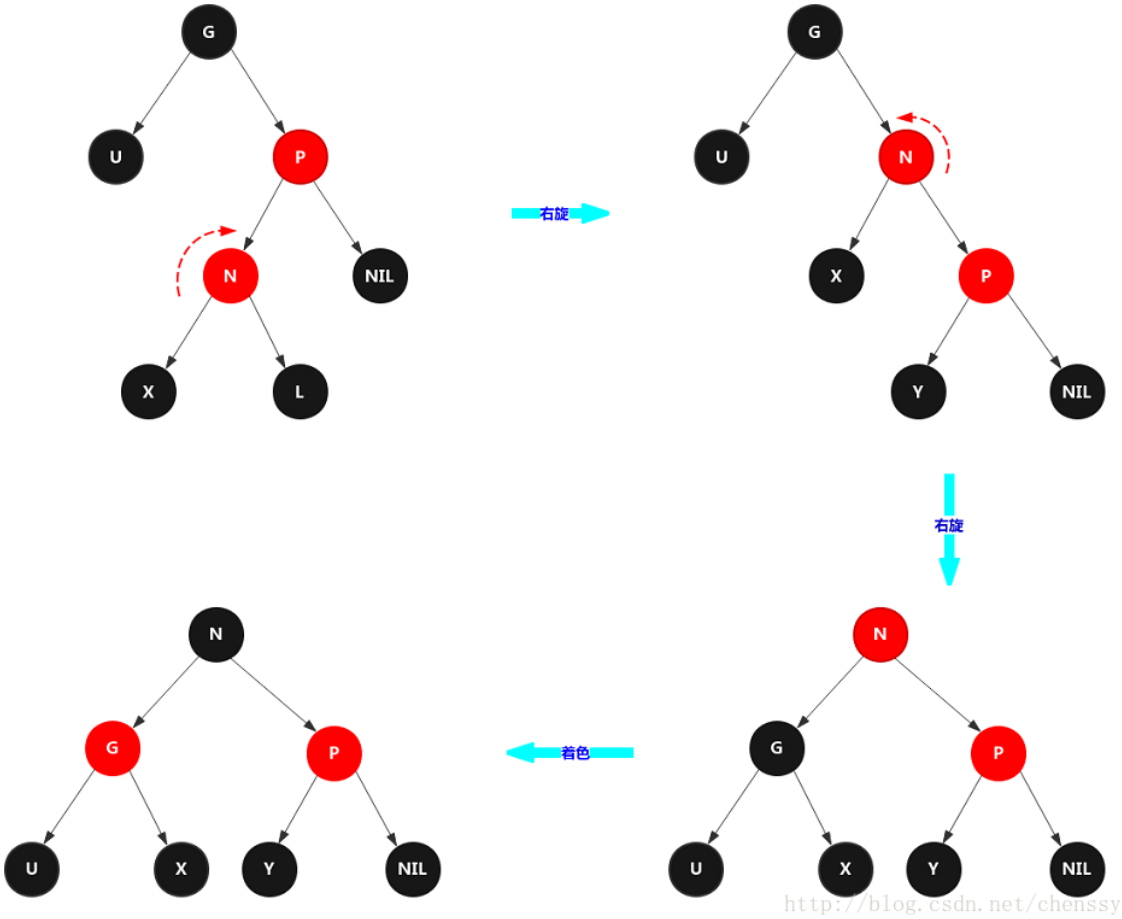
当然这样变色可能会导致另外一个问题了，就是父节点G与其父节点GG颜色冲突（上图二），那么这里需要将G节点当做新增节点进行递归处理。 **2 叔父U节点为黑叔** 如果当前节点的叔父节点U为黑色，则需要根据当前节点N与其父节点P的位置决定，分为四种情况：

1. N是P的右子节点、P是G的右子节点
2. N是P的左子节点，P是G的左子节点
3. N是P的左子节点，P是G的右子节点
4. N是P的右子节点，P是G的左子节点

情况1、2称之为外侧插入、情况3、4是内侧插入，之所以这样区分是因为他们的处理方式是相对的。 **2.1 外侧插入** 以N是P的右子节点、P是G的右子节点为例，这种情况的处理方式为：以P为支点进行左旋，然后交换P和G的颜色（P设置为黑色，G设置为红色），如下：



左外侧的情况（N是P的左子节点，P是G的左子节点）和上面的处理方式一样，先右旋，然后重新着色。 **2.2 内侧插入** 以N是P的左子节点，P是G的右子节点情况为例。内侧插入的情况稍微复杂些，经过一次旋转、着色是无法调整为红黑树的，处理方法如下：先进行一次右旋，再进行一次左旋，然后重新着色，即可完成调整。注意这里两次右旋都是以新增节点N为支点不是P。这里将N节点的两个NIL节点命名为X、L。如下：



至于左内侧则处理逻辑如下：先进行右旋，然后左旋，最后着色。

ConcurrentHashMap 的treeifyBin过程

ConcurrentHashMap的链表转换为红黑树过程就是一个红黑树增加节点的过程。在put过程中，如果发现链表结构中的元素超过了TREEIFY_THRESHOLD（默认为8），则会把链表转换为红黑树：

```
if (binCount >= TREEIFY_THRESHOLD)
    treeifyBin(tab, i);
```

treeifyBin主要的功能就是把链表所有的节点Node转换为TreeNode节点，如下：

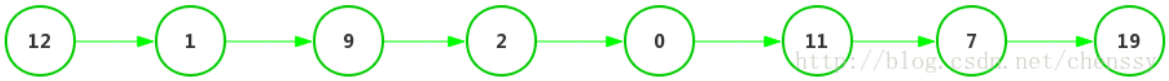
```
    private final void treeifyBin(Node<K,V>[] tab, int index) {
        Node<K,V> b; int n, sc;
        if (tab != null) {
            if ((n = tab.length) < MIN_TREEIFY_CAPACITY)
                tryPresize(n << 1);
            else if ((b = tabAt(tab, index)) != null && b.hash >= 0) {
                synchronized (b) {
                    if (tabAt(tab, index) == b) {
                        TreeNode<K,V> hd = null, tl = null;
                        for (Node<K,V> e = b; e != null; e = e.next) {
                            TreeNode<K,V> p =
                                new TreeNode<K,V>(e.hash, e.key, e.val,
                                                  null, null);
                            if ((p.prev = tl) == null)
                                hd = p;
                            else
                                tl.next = p;
                            tl = p;
```

```
                        }
                        setTabAt(tab, index, new TreeBin<K,V>(hd));
                    }
                }
            }
        }
    }
```

先判断当前Node的数组长度是否小于MIN_TREEIFY_CAPACITY（64），如果小于则调用tryPresize扩容处理以缓解单个链表元素过大的性能问题。否则则将Node节点的链表转换为TreeNode的节点链表，构建完成之后调用setTabAt()构建红黑树。TreeNode继承Node，如下：

```
    static final class TreeNode<K,V> extends Node<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;

        TreeNode(int hash, K key, V val, Node<K,V> next,
                 TreeNode<K,V> parent) {
            super(hash, key, val, next);
            this.parent = parent;
        }
        ......
    }
```

我们以下面一个链表作为案例，结合源代码来分析ConcurrentHashMap创建红黑树的过程：



12 12作为跟节点，直接为将红编程黑即可，对应源码：

```
                next = (TreeNode<K,V>)x.next;
                x.left = x.right = null;
                if (r == null) {
                    x.parent = null;
                    x.red = false;
                    r = x;
                }
```

12（**【注】：为了方便起见，这里省略NIL节点，后面也一样**）**1** 此时根节点root不为空，则插入节点时需要找到合适的插入位置，源码如下：

```
                    K k = x.key;
                    int h = x.hash;
                    Class<?> kc = null;
                    for (TreeNode<K,V> p = r;;) {
                        int dir, ph;
                        K pk = p.key;
                        if ((ph = p.hash) > h)
                            dir = -1;
                        else if (ph < h)
```

```
                        dir = 1;
                    else if ((kc == null &&
                              (kc = comparableClassFor(k)) == null) ||
                             (dir = compareComparables(kc, k, pk)) == 0)
                        dir = tieBreakOrder(k, pk);
                        TreeNode<K,V> xp = p;
                    if ((p = (dir <= 0) ? p.left : p.right) == null) {
                        x.parent = xp;
                        if (dir <= 0)
                            xp.left = x;
                        else
                            xp.right = x;
                        r = balanceInsertion(r, x);
                        break;
                    }
                }
```

从上面可以看到起处理逻辑如下：

1. 计算节点的hash值 p。dir 表示为往左移还是往右移。x 表示要插入的节点，p 表示带比较的节点。
2. 从根节点出发，计算比较节点p的的hash值 ph ，若ph > h ,则dir = -1 ,表示左移，取p = p.left。若 p == null则插入，若 p != null，则继续比较，直到直到一个合适的位置，最后调用 balanceInsertion()方法调整红黑树结构。ph < h，右移。
3. 如果ph = h，则表示节点“冲突”（和HashMap冲突一致），那怎么处理呢？首先调用 comparableClassFor()方法判断节点的key是否实现了Comparable接口，如果kc ！= null ，则通过compareComparables()方法通过compareTo()比较带下，如果还是返回 0，即dir == 0，则调用tieBreakOrder()方法来比较了。tieBreakOrder如下：

```
    static int tieBreakOrder(Object a, Object b) {
        int d;
        if (a == null || b == null ||
            (d = a.getClass().getName().
             compareTo(b.getClass().getName())) == 0)
            d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
                 -1 : 1);
        return d;
    }
```

tieBreakOrder()方法最终还是通过调用System.identityHashCode()方法来比较。 确定插入位置后，插入，由于插入的节点有可能会打破红黑树的结构，所以插入后调用balanceInsertion()方法来调整红黑树结构。

```
    static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,
                                                TreeNode<K,V> x) {
        x.red = true;       // 所有节点默认插入为红
        for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {

            // x.parent == null，为跟节点，置黑即可
            if ((xp = x.parent) == null) {
                x.red = false;
                return x;
            }
            // x 父节点为黑色，或者x 的祖父节点为空，直接插入返回
            else if (!xp.red || (xpp = xp.parent) == null)
```

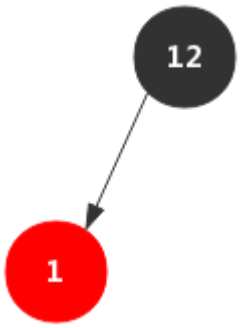
```
                return root;

            /*
             * x 的 父节点为红色
             * ----------------------
             * x 的 父节点 为 其祖父节点的左子节点
             */
            if (xp == (xppl = xpp.left)) {
                /*
                 * x的叔父节点存在，且为红色，颜色交换即可
                 * x的父节点、叔父节点变为黑色，祖父节点变为红色
                 */
                if ((xppr = xpp.right) != null && xppr.red) {
                    xppr.red = false;
                    xp.red = false;
                    xpp.red = true;
                    x = xpp;
                }
                else {
                    /*
                     * x 为 其父节点的右子节点，则为内侧插入
                     * 则先左旋，然后右旋
                     */
                    if (x == xp.right) {
                        // 左旋
                        root = rotateLeft(root, x = xp);
                        // 左旋之后x则会变成xp的父节点
                        xpp = (xp = x.parent) == null ? null : xp.parent;
                    }

                    /**
                     * 这里有两部分。
                     * 第一部分：x 原本就是其父节点的左子节点，则为外侧插入，右旋即可
                     * 第二部分：内侧插入后，先进行左旋，然后右旋
                     */
                    if (xp != null) {
                        xp.red = false;
                        if (xpp != null) {
                            xpp.red = true;
                            root = rotateRight(root, xpp);
                        }
                    }
                }
            }

            /**
             * 与上相对应
             */
            else {
                if (xppl != null && xppl.red) {
                    xppl.red = false;
                    xp.red = false;
                    xpp.red = true;
                    x = xpp;
                }
                else {
                    if (x == xp.left) {
                        root = rotateRight(root, x = xp);
```

```
                        xpp = (xp = x.parent) == null ? null : xp.parent;
                    }
                    if (xp != null) {
                        xp.red = false;
                        if (xpp != null) {
                            xpp.red = true;
                            root = rotateLeft(root, xpp);
                        }
                    }
                }
            }
        }
    }
```

回到节点1，其父节点为黑色，即：

```
            else if (!xp.red || (xpp = xp.parent) == null)
                return root;
```

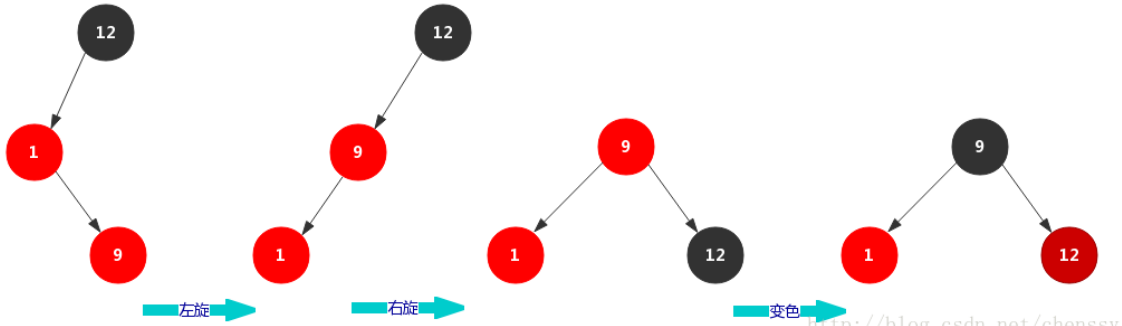
直接插入：



9 9作为1的右子节点插入，但是存在红红冲突，此时9的并没有叔父节点。9的父节点1为12的左子节点，9为其父节点1的右子节点，所以处理逻辑是先左旋，然后右旋，对应代码如下：

```
                    if (x == xp.right) {
                        root = rotateLeft(root, x = xp);
                        xpp = (xp = x.parent) == null ? null : xp.parent;
                    }
                    if (xp != null) {
                        xp.red = false;
                        if (xpp != null) {
                            xpp.red = true;
                            root = rotateRight(root, xpp);
                        }
                    }
```

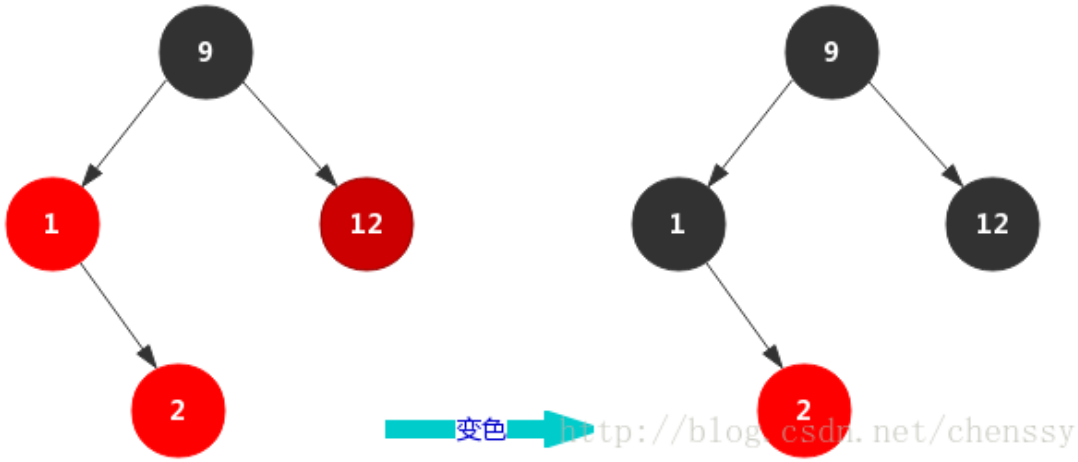
图例变化如下：



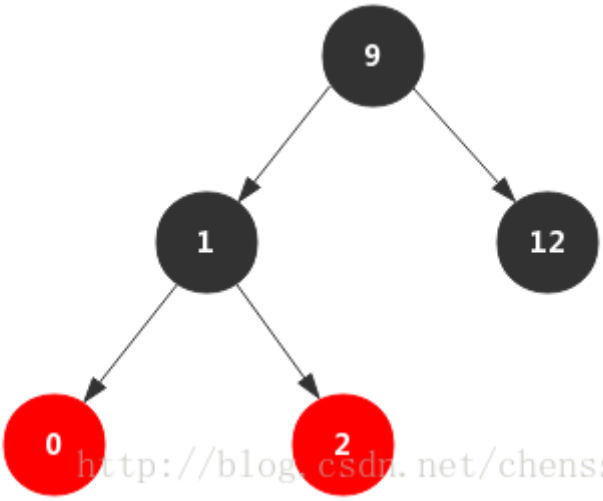
2 节点2 作为1 的右子节点插入，红红冲突，切其叔父节点为红色，直接变色即可，：

```
                    if ((xppr = xpp.right) != null && xppr.red) {
                        xppr.red = false;
                        xp.red = false;
                        xpp.red = true;
                        x = xpp;
                    }
```

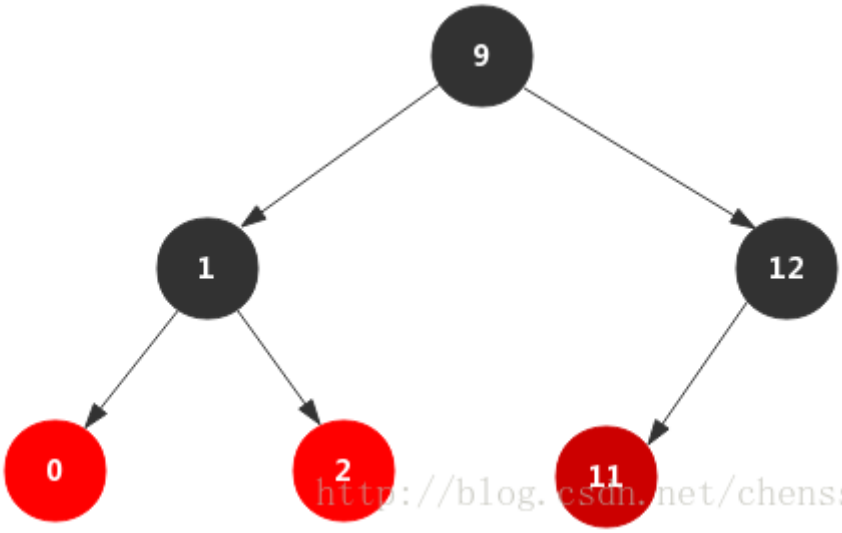
对应图例为：



0 节点0作为1的左子节点插入，由于其父节点为黑色，不会插入后不会打破红黑树结构，直接插入即可：

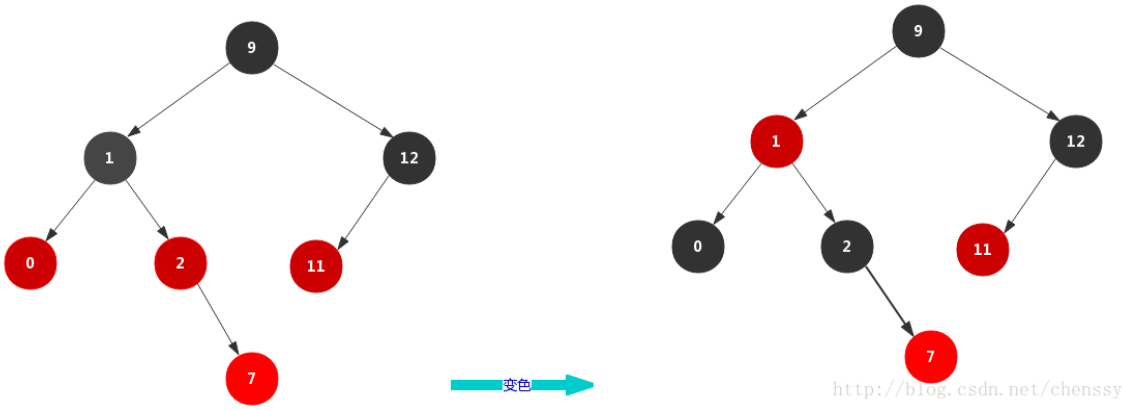


11 节点11作为12的左子节点，其父节点12为黑色，和0一样道理，直接插入：

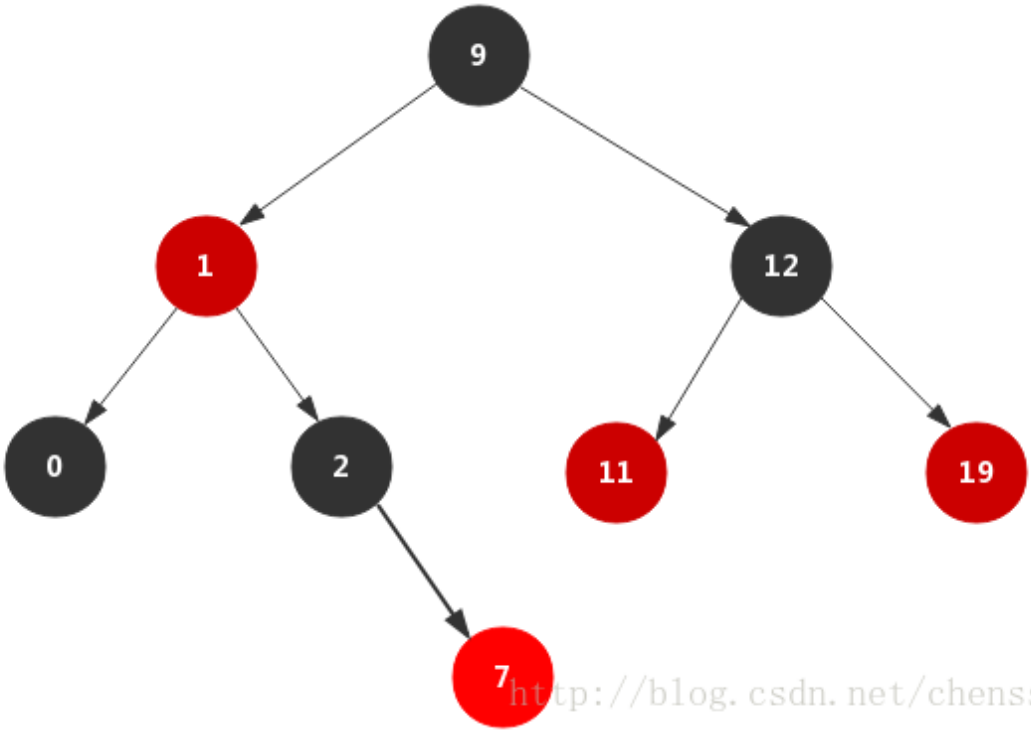


7 节点7作为2右

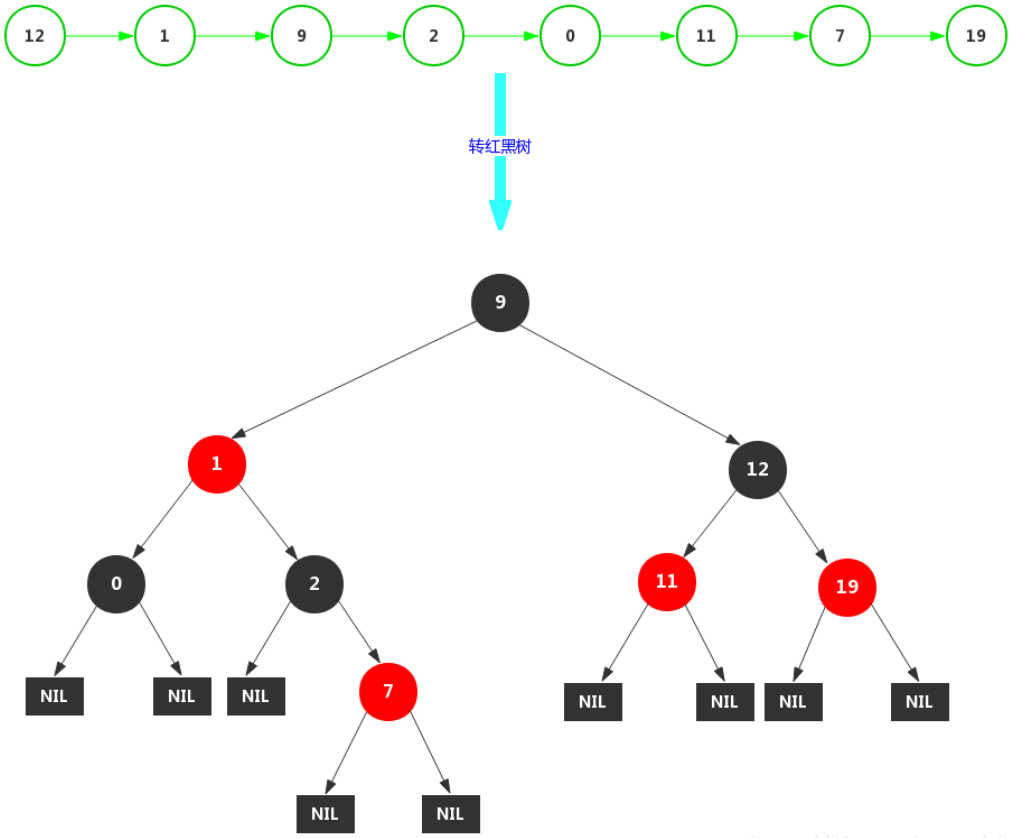
子节点插入，红红冲突，其叔父节点0为红色，变色即可：



19 节点19作为节点12的右子节点，直接插入：



至此，整个过程已经完成了，最终结果如下：

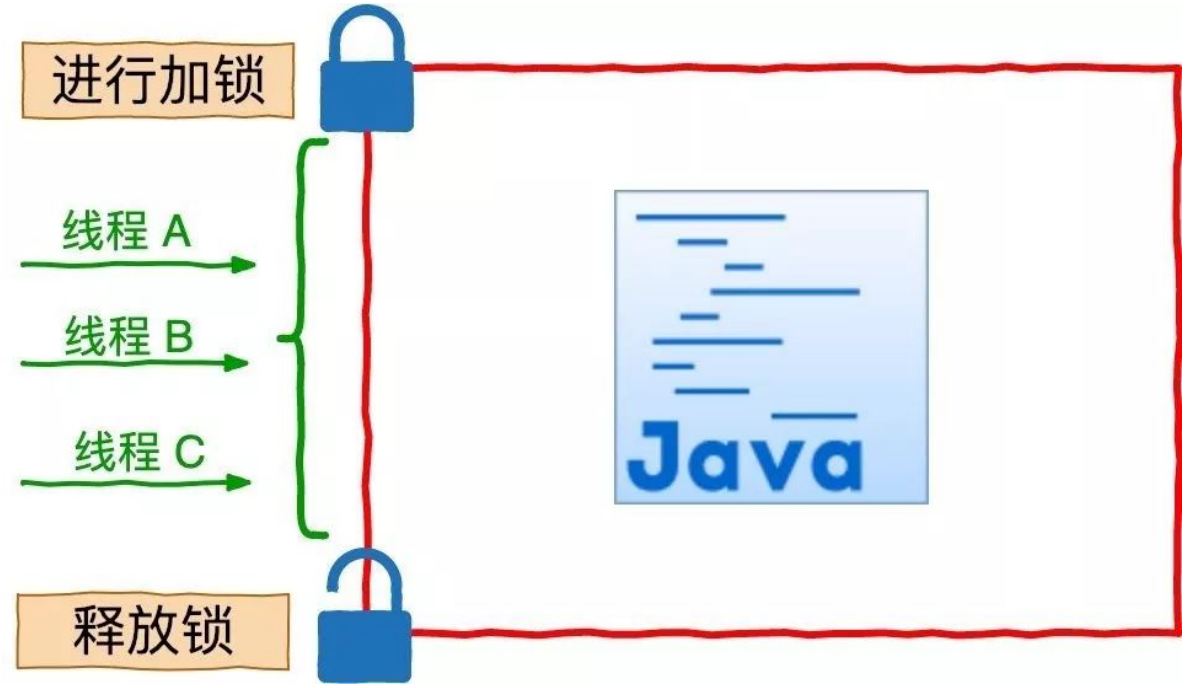


CAS原子操作及相关类

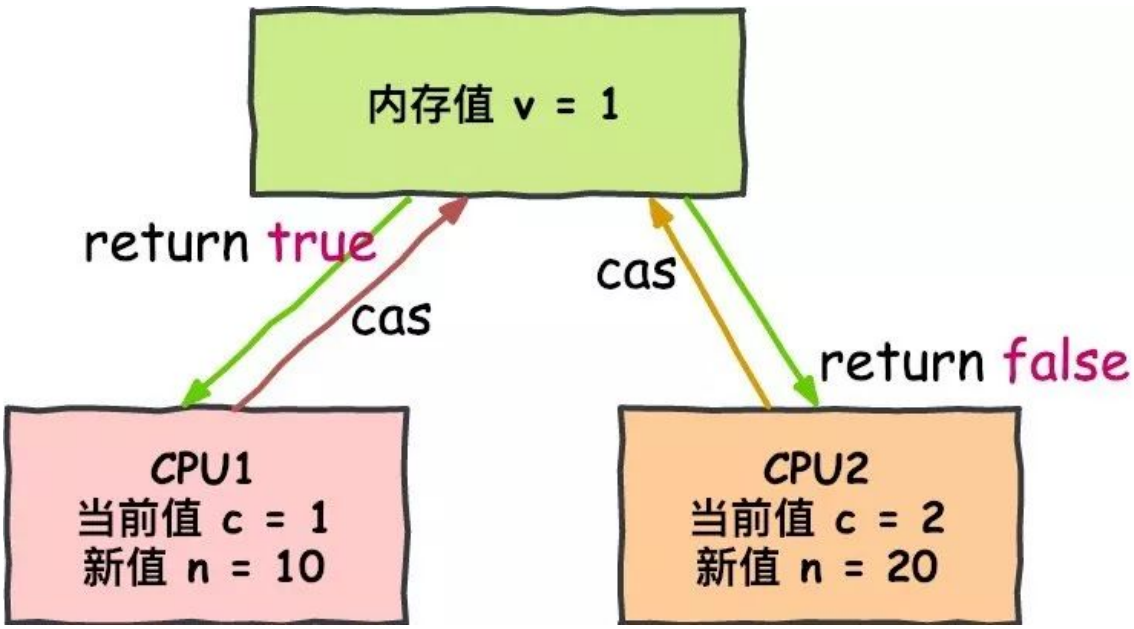
后端开发中大家肯定遇到过实现一个线程安全的计数器这种需求，根据经验你应该知道我们要在多线程中实现 **共享变量** 的原子性和可见性问题，于是锁成为一个不可避免的话题，今天我们讨论的是与之对应的无锁 CAS。本文会从怎么来的、是什么、怎么用、原理分析、遇到的问题等不同的角度带你真正搞懂 CAS。

为什么要无锁

我们一想到在多线程下保证安全的方式头一个要拎出来的肯定是锁，不管从硬件、操作系统层面都或多或少在使用锁。锁有什么缺点吗？当然有了，不然 JDK 里为什么出现那么多各式各样的锁，就是因为每一种锁都有其优劣势。



使用锁就需要获得锁、释放锁，CPU 需要通过上下文切换和调度管理来进行这个操作，对于一个 **独占锁** 而言一个线程在持有锁后没执行结束其他的哥们就必须在外面等着，等到前面的哥们执行完毕 CPU 大哥就会把锁拿出来其他的线程来抢了（非公平）。锁的这种概念基于一种悲观机制，它总是认为数据会被修改，所以你在操作一部分代码块之前先加一把锁，操作完毕后再释放，这样就安全了。其实在 JDK1.5 使用 `synchronized` 就可以做到。



但是像上面的操作在多线程下会让 CPU 不断的切换，非常消耗资源，我们知道可以使用具体的某一类锁来避免部分问题。那除了锁的方式还有其他的吗？当然，有人就提出了无锁算法，比较有名的就是我们今天要说的 CAS（compare and swap），和锁不同的是它是一种乐观的机制，它认为别人去拿数据的时候不会修改，但是在修改数据的时候去判断一下数据此时的状态，这样的话 CPU 不会切换，在读多的情况下性能将得到大幅提升。当前我们使用的大部分 CPU 都有 CAS 指令了，从硬件层面支持无锁，这样开发的时候去调用就可以了。

不论是锁还是无锁都有其优劣势，后面我们也会通过例子说明 CAS 的问题。

什么是 CAS

前面提了无锁的 CAS，那到底 CAS 是个啥呢？我已经迫不及待了，我们来看看维基百科的解释

> 比较并交换(compare and swap, CAS)，是原子操作的一种，可用于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某一数据时由于执行顺序不确定性以及中断的不可预知性产生的数据不一致问题。该操作通过将内存中的值与指定数据进行比较，当数值一样时将内存中的数据替换为新的值。

CAS 给我们提供了一种思路，通过 **比较** 和 **替换** 来完成原子性，来看一段代码：

```
int cas(long *addr, long old, long new) {
    /* 原子执行 */
    if(*addr != old)
        return 0;
    *addr = new;
    return 1;
}
```

这是一段 c 语言代码，可以看到有 3 个参数，分别是：

- `*addr`: 进行比较的值
- `old`: 内存当前值
- `new`: 准备修改的新值，写入到内存

只要我们当前传入的进行比较的值和内存里的值相等，就将新值修改成功，否则返回 0 告诉比较失败了。学过数据库的同学都知道悲观锁和乐观锁，乐观锁总是认为数据不会被修改。基于这种假设 CAS 的操作也认为内存里的值和当前值是相等的，所以操作总是能成功，我们可以不需要加锁就实现多线程下的原子性操作。

在多线程情况下使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被阻塞挂起，而是告诉它这次修改失败了，你可以重新尝试，于是可以写这样的代码。

```
while (!cas(&addr, old, newValue)) {

}
// success
printf("new value = %ld", addr);
```

不过这样的代码相信你可能看出其中的蹊跷了，这个我们后面来分析，下面来看看 Java 里是怎么用 CAS 的。

Java 里的 CAS

还是前面的问题，如果让你用 Java 的 API 来实现你可能会想到两种方式，一种是加锁（可能是 synchronized 或者其他种类的锁），另一种是使用 `atomic` 类，如 `AtomicInteger`，这一系列类是在 JDK1.5 的时候出现的，在我们常用的 `java.util.concurrent.atomic` 包下，我们来看个例子：

```
ExecutorService executorService = Executors.newCachedThreadPool();
AtomicInteger   atomicInteger   = new AtomicInteger(0);

for (int i = 0; i < 5000; i++) {
    executorService.execute(atomicInteger::incrementAndGet);
}

System.out.println(atomicInteger.get());
executorService.shutdown();
```

这个例子开启了 5000 个线程去进行累加操作，不管你执行多少次答案都是 5000。这么神奇的操作是如何实现的呢？就是依靠 CAS 这种技术来完成的，我们揭开 `AtomicInteger` 的老底看看它的代码：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

    /**
     * Creates a new AtomicInteger with the given initial value.
     *
     * @param initialValue the initial value
     */
    public AtomicInteger(int initialValue) {
        value = initialValue;
    }

    /**
     * Gets the current value.
     *
     * @return the current value
     */
    public final int get() {
        return value;
    }

    /**
     * Atomically increments by one the current value.
     *
     * @return the updated value
     */
    public final int incrementAndGet() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }
```

```
}
```

这里我只帖出了我们前面例子相关的代码，其他都是类似的，可以看到 `incrementAndGet` 调用了 `unsafe.getAndAddInt` 方法。`Unsafe` 这个类是 JDK 提供的一个比较底层的类，它不让我们程序员直接使用，主要是怕操作不当把机器玩坏了。。。（其实可以通过反射的方式获取到这个类的实例）你会在 JDK 源码的很多地方看到这家伙，我们先说说它有什么能力：

- 内存管理：包括分配内存、释放内存
- 操作类、对象、变量：通过获取对象和变量偏移量直接修改数据
- 挂起与恢复：将线程阻塞或者恢复阻塞状态
- CAS：调用 CPU 的 CAS 指令进行比较和交换
- 内存屏障：定义内存屏障，避免指令重排序

这里只是大致提一下常用的操作，具体细节可以在文末的参考链接中查看。下面我们继续看 `unsafe` 的 `getAndAddInt` 在做什么。

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

public native int getIntVolatile(Object var1, long var2);
public final native boolean compareAndSwapInt(Object var1, long var2, int var4,
int var5);
```

其实很简单，先通过 `getIntVolatile` 获取到内存的当前值，然后进行比较，展开 `compareAndSwapInt` 方法的几个参数：

- `var1`: 当前要操作的对象（其实就是 `AtomicInteger` 实例）
- `var2`: 当前要操作的变量偏移量（可以理解为 CAS 中的内存当前值）
- `var4`: 期望内存中的值
- `var5`: 要修改的新值

所以 `this.compareAndSwapInt(var1, var2, var5, var5 + var4)` 的意思就是，比较一下 `var2` 和内存当前值 `var5` 是否相等，如果相等那我就将内存值 `var5` 修改为 `var5 + var4`（`var4` 就是 1，也可以是其他数）。

这里我们还需要解释一下 **偏移量** 是个啥？你在前面的代码中可能看到这么一段：

```
// setup to use Unsafe.compareAndSwapInt for updates
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;
static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
    }
private volatile int value;
```

可以看出在静态代码块执行的时候将 `AtomicInteger` 类的 `value` 这个字段的偏移量获取出来，拿这个 long 数据干嘛呢？在 `Unsafe` 类里很多地方都需要传入 `obj` 和偏移量，结合我们说 `Unsafe` 的诸多能力，其实就是直接通过更底层的方式将对象字段在内存的数据修改掉。

使用上面的方式就可以很好的解决多线程下的原子性和可见性问题。由于代码里使用了 `do while` 这种循环结构，所以 CPU 不会被挂起，比较失败后重试，就不存在上下文切换了，实现了无锁并发编程。

CAS 存在的问题

自旋的劣势

你留意上面的代码会发现一个问题，`while` 循环如果在最坏情况下总是失败怎么办？会导致 CPU 在不断处理。像这种 `while(!compareAndSwapInt)` 的操作我们称之为自旋，CAS 是乐观的，认为大家来并不都是修改数据的，现实可能出现非常多的线程过来都要修改这个数据，此时随着并发量的增加会导致 CAS 操作长时间不成功，CPU 也会有很大的开销。所以我们要清楚，如果是读多写少的情况也就满足乐观，性能是非常好的。

ABA 问题

提到 CAS 不得不说 ABA 问题，它是说假如内存的值原来是 A，被一个线程修改为了 B，此时又有一个线程把它修改为了 A，那么 CAS 肯定是操作成功的。真的这样做的话代码可能就有 bug 了，对于修改数据为 B 的那个线程它应该读取到 B 而不是 A，如果你做过数据库相关的乐观锁机制可能会想到我们在比较的时候使用一个版本号 `version` 来进行判断就可以搞定。在 JDK 里提供了一个 `AtomicStampedReference` 类来解决这个问题，来看一个例子：

```
int stamp = 10001;
AtomicStampedReference<Integer> stampedReference = new AtomicStampedReference<>
(0, stamp);
stampedReference.compareAndSet(0, 10, stamp, stamp + 1);
System.out.println("value: " + stampedReference.getReference());
System.out.println("stamp: " + stampedReference.getStamp());
```

它的构造函数是 2 个参数，多传入了一个初始 时间戳，用这个戳来给数据加了一个版本，这样的话多个线程来修改如果提供的戳不同。在修改数据的时候除了提供一个新的值之外还要提供一个新的戳，这样在多线程情况下只要数据被修改了那么戳一定会发生改变，另一个线程拿到的是旧的戳所以会修改失败。

尝试应用

既然 CAS 提供了这么好的 API，我们不妨用它来实现一个简易版的独占锁。思路是当某个线程进入 `lock` 方法就比较锁对象的内存值是否是 false，如果是则代表这把锁它可以获取，获取后将内存之修改为 true，获取不到就自旋。在 `unlock` 的时候将内存值再修改为 false 即可，代码如下：

```
 public class SpinLock {

     private AtomicBoolean mutex = new AtomicBoolean(false);

     public void lock() {
         while (!mutex.compareAndSet(false, true)) {
             // System.out.println(Thread.currentThread().getName()+ " wait lock
release");
         }
```

```
     }

    public void unlock() {
        while (!mutex.compareAndSet(true, false)) {
            // System.out.println(Thread.currentThread().getName()+ " wait lock
release");
        }
    }

}
```

这里使用了 `AtomicBoolean` 这个类，当然用 `AtomicInteger` 也是可以的，因为我们只保存一个状态 `boolean` 占用比较小就用它了。这个锁的实现比较简单，缺点非常明显，由于 `while` 循环导致的自旋会让其他线程都在占用 CPU，但是也可以使用，关于锁的优化版本实现我会在后续的文章中进行改进和说明，正因为这些问题我们也会在后续研究 `AQS` 这把利器的优点。

CAS 源码

看了上面的这些代码和解释相信你对 CAS 已经理解了，下面我们要说的原理是前面的 `native` 方法中的 C++ 代码写了什么，在 openjdk 的 `/hotspot/src/share/vm/prims` 目录中有一个 Unsafe.cpp 文件中有这样一段代码：

> 注意：这里以 hotspot 实现为例

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe,
jobject obj, jlong offset, jint e, jint x))
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  // 通过偏移量获取对象变量地址
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  // 执行一个原子操作
  // 如果结果和现在不同，就直接返回，因为有其他人修改了；否则会一直尝试去修改。直到成功。
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```

AtomicInteger原理分析

在多线程的场景中，我们需要保证数据安全，就会考虑同步的方案，通常会使用synchronized或者lock来处理，使用了synchronized意味着内核态的一次切换。这是一个很重的操作。

有没有一种方式，可以比较便利的实现一些简单的数据同步，比如计数器等等。concurrent包下的atomic提供我们这么一种轻量级的数据同步的选择。

使用例子

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class App {

    public static void main(String[] args) throws Exception {
        CountDownLatch countDownLatch = new CountDownLatch(100);

        AtomicInteger atomicInteger = new AtomicInteger(0);
```

```
        for (int i = 0; i < 100; i++) {
            new Thread() {
                @Override
                public void run() {
                    atomicInteger.getAndIncrement();

                    countDownLatch.countDown();
                }
            }.start();
        }

        countDownLatch.await();

        System.out.println(atomicInteger.get());
    }
}
```

在以上代码中，使用AtomicInteger声明了一个全局变量，并且在多线程中进行自增，代码中并没有进行显示的加锁。

以上代码的输出结果，永远都是100。如果将AtomicInteger换成Integer，打印结果基本都是小于100。

也就说明AtomicInteger声明的变量，在多线程场景中的自增操作是可以保证线程安全的。接下来我们分析下其原理。

原理

我们可以看一下AtomicInteger的代码

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
        //存在一个volatile的int里面。volatile只能保证这个变量的可见性。不能保证他的原子性。
    private volatile int value;

    /**
     * Creates a new AtomicInteger with the given initial value.
     *
     * @param initialValue the initial value
     */
    public AtomicInteger(int initialValue) {

        value = initialValue;
    }
    //可以看看getAndIncrement这个类似i++的函数，可以发现，是调用了UnSafe中的getAndAddInt。
```

```
    public final int getAndIncrement() {
        return unsafe.getAndAddInt(this, valueOffset, 1);
    }
```

UnSafe提供了java可以直接操作底层的能力。

进一步，我们可以发现实现方式：

如何保证原子性：**自旋 + CAS（乐观锁）**。在这个过程中，通过compareAndSwapInt比较更新value值，如果更新失败，重新获取旧值，然后更新。

```
public final class Unsafe {

  public final int getAndAddInt(Object var1, long var2, int var4) {
      int var5;
      do {
          var5 = this.getIntVolatile(var1, var2);
      } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

      return var5;
  }
}
```

CAS相对于其他锁，不会进行内核态操作，有着一些性能的提升。但同时引入自旋，当锁竞争较大的时候，自旋次数会增多。cpu资源会消耗很高。

换句话说，CAS+自旋适合使用在低并发有同步数据的应用场景。

Java 8做出的改进

在Java 8中引入了4个新的计数器类型，LongAdder、LongAccumulator、DoubleAdder、DoubleAccumulator。他们都是继承于Striped64。

在LongAdder 与AtomicLong有什么区别？
Atomic*遇到的问题是，只能运用于低并发场景。因此LongAddr在这基础上引入了分段锁的概念。可以参考《JDK8系列之LongAdder解析》一起看看做了什么。

大概就是当竞争不激烈的时候，所有线程都是通过CAS对同一个变量（Base）进行修改，当竞争激烈的时候，会将根据当前线程哈希到对于Cell上进行修改（多段锁）。

```
abstract class Striped64 extends Number {
  /**
   * Padded variant of AtomicLong supporting only raw accesses plus CAS.
   *
   * JVM intrinsics note: It would be possible to use a release-only
   * form of CAS here, if it were provided.
   */
  @sun.misc.Contended static final class Cell {
      volatile long value;
      Cell(long x) { value = x; }
      final boolean cas(long cmp, long val) {
          return UNSAFE.compareAndSwapLong(this, valueOffset, cmp, val);
      }

      // Unsafe mechanics
      private static final sun.misc.Unsafe UNSAFE;
      private static final long valueOffset;
```

```
        static {
            try {
                UNSAFE = sun.misc.Unsafe.getUnsafe();
                Class<?> ak = Cell.class;
                valueOffset = UNSAFE.objectFieldOffset
                    (ak.getDeclaredField("value"));
            } catch (Exception e) {
                throw new Error(e);
            }
        }
    }

    /** Number of CPUS, to place bound on table size */
    static final int NCPU = Runtime.getRuntime().availableProcessors();

    /**
     * Table of cells. When non-null, size is a power of 2.
     */
    transient volatile Cell[] cells;

    /**
     * Base value, used mainly when there is no contention, but also as
     * a fallback during table initialization races. Updated via CAS.
     */
    transient volatile long base;
}
```

参考资料

- cas wiki

  https://zh.wikipedia.org/wiki/%E6%AF%94%E8%BE%83%E5%B9%B6%E4%BA%A4%E6%8D%A2

- 说一说Java的Unsafe类

  https://www.cnblogs.com/pkufork/p/java_unsafe.html

- Java Magic. Part 4: sun.misc.Unsafe

  http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/